CASA Next Generation Infrastructure Documentation

Release 0.1b

National Radio Astronomy Observatory

Sep 03, 2021

CONTENTS

A project to replace the MeasurementSet and Image data manipulation and processing infrastructure of CASA with a parallel, scalable, and largely off-the-shelf solution.

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/introduction.ipynb

CHAPTER

ONE

INTRODUCTION

CASA software development faces several competing challenges that are expected to only worsen with time. As both a scientific research and development platform as well as a necessary component in multi-million dollar telescope operations, CASA must overcome the contradictory goals of being a flexible experimentation platform with mature, documented, tested, and stable performance.

Furthermore, it must accomplish this with a relatively modest development staff that is diverse, remote, and matrixed across many competing projects.

Historically, much of CASA traces its roots to algorithms developed and refined over many decades and implemented in a custom code base utilized and maintained by a global, yet small developer base. In years past, it was necessary to develop a custom infrastructure to support the unique needs of radio astronomy science. Yet as data sizes, performance demands, and conflicting needs of multiple telescopes continue to rise, the overhead of maintaining this infrastructure grows exponentially.

The CASA team has begun exploring options for a new generation of software to meet the growing demands of current and future instruments. This **cngi_prototype** package is a demonstration of the current state of our research efforts. Its primary purpose is to showcase new *data structures* for MeasurementSet and Image contents built entirely in Python atop the popular technology stack of numpy, dask, and xarray. A selection of core mathematics, manipulation, middleware and analysis functions are shown in the *Visibility* and *Image* overview sections to demonstrate the simplicity and scalability of the technology choices. Notional examples of *Calibration*, *Flagging* and *Imaging* are provided to illustrate future design and implementation direction. Finally, the most compute intensive areas of CASA imaging are implemented and *benchmarked* to demonstrate the parallel scalability and raw performance now possible from a pure-Python software stack.

A detailed explanation of technology choices, including the xarray and dask frameworks, the zarr storage format, and the functional design architecture can be found in the *Development* section.

1.1 Project Background

The Common Astronomy Support Applications (CASA) package supports a variety of radio astronomy needs by providing mechanisms for data manipulation, analysis, calibration and imaging. Derivatives of these broad categories include items such as visualization, simulation, conversion, and experimentation. The primary data products of CASA are the MeasurementSet (MS) and Image file formats. Additional products are not discussed here but include calibration tables and imaging cache.

CASA is a layered collection of libraries, classes and functions spanning both C++ and Python languages as well as Object Oriented and Functional paradigms. The base layer of CASA includes a standalone project, known as casacore, developed and maintained here: https://github.com/casacore/casacore, as well as various external libraries to support mathematics, parallelization, visualization, etc. The base layer is wrapped under a middleware layer of CASA that abstracts some of the details of data access and allows for simpler expression of science algorithms. The top layers are user-facing tools and tasks that directly expose the science of radio interferometer data reduction.

1.1.1 Goals

A great deal of the engineering complexity and processing time in CASA stems from the way in which underlying data is accessed and manipulated in the infrastructure layers. By overhauling these base infrastructure layers, significant savings in complexity and processing time along with dramatic improvements to science development flexibility and potential can be realized. Here we highlight the main goals and opportunities for improvement within the CASA codebase:

Maximize Performance

- Fast experimentation
- (Near) linear scalability allowing users to reduce processing time by adding cheap commodity hardware resources
- Extreme scalability to thousands of cores and petascale data sizes needed by next generation instruments such as ngVLA
- Fully utilize all cores, memory, and disk I/O simultaneously
- Support everything from single user laptops to high-end clusters in enterprise datacenters as well as fully distributed grid and cloud processing environments

Minimize engineering overhead

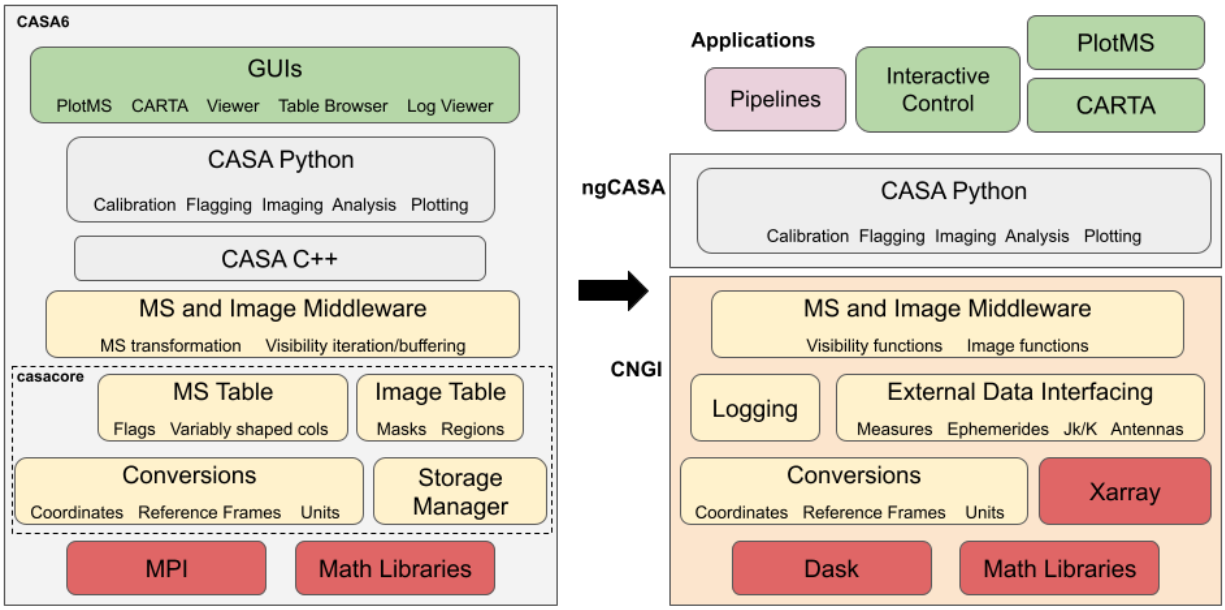
- Simplify scientific implementations without distraction of software engineering complexity
- Reduce custom code base by an order of magnitude
- Reduce development time of new features and maintenance overhead of existing features
- Support a variety of use cases and interoperability with other packages by leveraging off-the-shelf standards, tools, and formats

1.1.2 Scope

A complete top-down overhaul of the entirety of CASA is not feasible as a first (and only) step, as there are no complete and detailed requirements specifications, design documents, architecture or definition of correct output. Instead, the scope of this project is to first condense and replace the CASA data processing software infrastructure and casacore code base only with a new and functionally equivalent package (CNGI) in a bottom-up approach. As a consequence of this infrastructure replacement, a second effort to migrate and/or replace significant portions of CASA science functionality is planned as an additional follow-on project (ngCASA).

Although roughly analogous to the current separation of CASA and casacore, the new ngCASA / CNGI separation will include more capabilities in CNGI than existed in casacore. An additional distinction is being made between ngCASA as a library of Python functions, and the user applications built around those functions (i.e. PlotMS).

The following diagram illustrates the scope of CNGI and ngCASA, and the ramifications for subsequent development.



1.1.3 CASA Next Generation Infrastructure (CNGI)

The CASA Next Generation Infrastructure (CNGI) project will provide a replacement to the infrastructure and middleware layers of CASA using largely off-the-shelf products that are dramatically simpler to use and scalable to a variety of computing environments. This will take the form of a processing package that includes a programming language, API, and hardware interface drivers. This package will be functionally equivalent to the MeasurementSet and Image data manipulation capabilities of the base and middleware layers it replaces, but realized in an entirely new paradigm.

1.1.4 Future Next Generation CASA (ngCASA)

The future Next Generation CASA project will use visibility and image manipulation methods from CNGI to implement a data analysis package that replaces the scientific application layer of CASA. This package will provide a set of functions that may be used either as stand-alone building blocks or strung together in a series (internally forming a DAG) to implement operations such as synthesis calibration and image reconstruction. A user of ngCASA may choose to implement their own analysis task, use a pre-packaged task similar to one in current CASA, or embed ngCASA methods in a custom pipeline DAG.

1.1.5 High Level Separation

The main purpose behind separating layers of the code base into individual CNGI and ngCASA packages (and then further separating ngCASA with an additional Application layer) is to better support a diverse and wide ranging user base. Global partnerships and collaborative agreements may be formed over the basic data types and processing engine in CNGI, with more observatory specific tailoring appearing the further up you go in the preceding diagram. An important but often overlooked distinction exists between the needs of interactive human users, automated pipelines, and other applications wishing to build in CASA capabilities. The addition of an application layer allows for better tailoring of the user experience to each different category of user.

1.1.6 Prototype Demonstration Package

This cngi_prototype demonstration package is a pilot effort to assess the technology choices made to date for the CNGI Project. It is primarily focused on prototyping CNGI-layer functionality to see how well it can work and perform against the goals of the project. A second objective is to show how technology choices are likely to satisfy future scientific and engineering needs of the ngCASA project. As such, some preliminary ngCASA layer components are included to build confidence that the infrastructure framework can handle the performance and complexity demands, and to illustrate how such functionality may look in the future.

A detailed explanation of the design and technology choices, including xarray, dask, zarr, data structure formats, and functional programming style can be found in the *Development* section.

A bottom-up strategy that begins with these prototype building blocks while emphasizing scalability, simplicity and flexibility ensures meaningful work can proceed in the absence of a detailed requirements specification of future ngCASA needs. This package is likely to form the starting point for full production development of CNGI and ngCASA at a later date. As such a high degree of code reuse is anticipated.

1.2 Installation

The CNGI_Prototype demonstration package is available as a pypi package for user installation and assessment. Note this is a demonstration prototype only and **not intended for science**.

This is a source distribution package, and should work on most recent OS and Python versions >= 3.6. However the functions to convert MS and Image data structures from current CASA format to the new format requires the CASA 6.2 casatools module. CASA 6 also requires FORTRAN libraries be installed in the OS. These are not included in the dependency list so that the rest of CNGI functionality may be used without these constraints.

In the future, the conversion module may be moved outside of the CNGI package and distributed separately.

1.2.1 Pip Installation

Mac Users may encounter an installation error with the python-casacore dependency. See the Conda installation method instead

```
bash$: python3 -m venv cngi
bash$: source cngi/bin/activate
(cngi) bash$: pip install --upgrade pip wheel
(cngi) bash$: pip install cngi-prototype
```

Sanity Check

```
(cngi) bash$: gdown --id 1ui6KxXivE0SRFW8na_ry1Blw-Zk5YDDH
(cngi) bash$: tar -xf M100.vis.zarr.tar
(cngi) bash$: python

>>> from cngi.dio import read_vis
>>>
>>> mxds = read_vis('M100.vis.zarr')
>>>
>>> print(mxds)
>>> print(mxds.xds0)
```

1.2.2 Pip Installation - Full CASA6 + CNGI

```
bash$: python3 -m venv cngi
bash$: source cngi/bin/activate
# [ubuntu only] sudo apt-get install libgfortran3
(cngi) bash$: pip install --upgrade pip wheel
(cngi) bash$: pip install casatools==6.2.0.124
(cngi) bash$: pip install casadata
(cngi) bash$: pip install cngi-prototype
```

Sanity check

```
(cngi) bash$: gdown --id 15HfB4rJKqEH7df088Ge5YLrCTXBIax6R
(cngi) bash$: tar -xf M100.ms.tar
(cngi) bash$: python

>>> from cngi.conversion import convert_ms
>>>
>>> mxds = convert_ms('M100.ms')
>>> print(mxds)
>>> print(mxds.xds0)
```

1.2.3 Conda Installation

```
bash$: conda create -n cngi python=3.6
bash$: conda activate cngi
bash$: conda install -c conda-forge python-casacore
bash$: pip install cngi-prototype
```

1.2.4 Installation from Source

```
bash$: git clone https://github.com/casangi/cngi_prototype.git
bash$: cd cngi_prototype
bash$: python3 -m venv venv
bash$: source venv/bin/activate
bash$: pip install --upgrade pip wheel
bash$: pip install -r requirements.txt
bash$: python setup.py install --root=.
```

1.2.5 Configure dask.distributed

To avoid thread collisions, when using the Dask.distributed Client, set the following environment variables.

```
export OMP_NUM_THREADS=1
export MKL_NUM_THREADS=1
export OPENBLAS_NUM_THREADS=1
```

1.2.6 Building Documentation from Source

Follow steps to install cngi from source. Then navigate to the docs folder and execute the following:

```
sphinx-build -b html . ./build
```

View the documentation in your browser by navigating to:

`sh file:///path/to/cngi/cngi_prototype/docs/build/index.html`

1.3 Usage

You can import things several different ways. For example:

```
>>> from cngi import dio
>>> xds = dio.read_image(...)
```

or

```
>>> from cngi.dio import read_image
>>> xds = read_image(...)
```

or

```
>>> import cngi.dio as cdio
>>> xds = cdio.read_image(...)
```

Throughout the documentation we use the variable name `xds` to refer to Xarray DataSets.
`xda` may be used to refer to Xarray DataArrays. This is a "loose" convention only.

CHAPTER

TWO

API

This package is composed of the following modules. Module functions with blank descriptions have not been completed yet. As each function is completed, its description will be populated.

2.1 cngi

CNGI Package

Module functions with blank descriptions have not been completed yet. As each function is completed, its description will be populated.

2.1.1 conversion

Legacy CASA uses a custom MS format while CNGI uses the standard Zarr format. These functions allow conversion between the two as well as directly from the telescope archival science data model (ASDM) (future growth). Note that both the MS and Zarr formats are directories, not single files.

This package has a dependency on legacy CASA / casacore and will be separated in the future to its own distribution apart from the rest of the CNGI package.

To access these functions, use your favorite variation of: `import cngi.conversion`

cngi.conversion.convert_image

convert_image(*infile*, *outfile=None*, *artifacts=[]*, *compressor=None*, *chunks=(- 1, - 1, 1, 1)*)

Convert legacy CASA or FITS format Image to xarray Image Dataset and zarr storage format

This function requires CASA6 casatools module.

Parameters

- **infile** (*str*) – Input image filename (.image or .fits format). If taylor terms are present, they should be in the form of filename.image.tt0 and this infile string should be filename.image
- **outfile** (*str*) – Output zarr filename. If None, will use infile name with .img.zarr extension
- **artifacts** (*list of str*) – List of other image artifacts to include if present with infile. Use None for just the specified infile. Default [] uses `['mask','model','pb','psf','residual','sumwt','weight']`

- **compressor** (*numcodecs.blosc.Blosc*) – The blosc compressor to use when saving the converted data to disk using zarr. If None the zstd compression algorithm used with compression level 2.
- **chunks** (*4-D tuple of ints*) – Shape of desired chunking in the form of (l, m, channels, polarization), use -1 for entire axis in one chunk. Default is (-1, -1, 1, 1) Note: chunk size is the product of the four numbers (up to the actual size of the dimension)

Returns new xarray Datasets of Image data contents

Return type xarray.core.dataset.Dataset

cngi.conversion.convert_ms

convert_ms(*infile*, *outfile=None*, *ddis=None*, *ignore=['HISTORY']*, *compressor=None*, *chunks=(100, 400, 32, 1)*, *sub_chunks=10000*, *append=False*)

Convert legacy format MS to xarray Visibility Dataset and zarr storage format

This function requires CASA6 casatools module. The CASA MSv2 format is converted to the MSv3 schema per the specified definition here: https://drive.google.com/file/d/10TZ4dsFw9CconBc-GFxSeb2caT6wkmza/view?usp=sharing

The MS is partitioned by DDI, which guarantees a fixed data shape per partition. This results in different subdirectories under the main vis.zarr folder. There is no DDI in MSv3, so this simply serves as a partition id in the zarr directory.

Parameters

- **infile** (*str*) – Input MS filename
- **outfile** (*str*) – Output zarr filename. If None, will use infile name with .vis.zarr extension
- **ddis** (*list*) – List of specific DDIs to convert. DDI's are integer values, or use 'global' string for subtables. Leave as None to convert entire MS
- **ignore** (*list*) – List of subtables to ignore (case sensitive and generally all uppercase). This is useful if a particular subtable is causing errors. Default is None. Note: default is now temporarily set to ignore the HISTORY table due a CASA6 issue in the table tool affecting a small set of test cases (set back to None if HISTORY is needed)
- **compressor** (*numcodecs.blosc.Blosc*) – The blosc compressor to use when saving the converted data to disk using zarr. If None the zstd compression algorithm used with compression level 2.
- **chunks** (*4-D tuple of ints*) – Shape of desired chunking in the form of (time, baseline, channel, polarization), use -1 for entire axis in one chunk. Default is (100, 400, 20, 1) Note: chunk size is the product of the four numbers, and data is batch processed by time axis, so that will drive memory needed for conversion.
- **sub_chunks** (*int*) – Chunking used for subtable conversion (except for POINTING which will use time/baseline dims from chunks parameter). This is a single integer used for the row-axis (d0) chunking only, no other dims in the subtables will be chunked.
- **append** (*bool*) – Keep destination zarr store intact and add new DDI's to it. Note that duplicate DDI's will still be overwritten. Default False deletes and replaces entire directory.

Returns Master xarray dataset of datasets for this visibility set

Return type xarray.core.dataset.Dataset

cngi.conversion.convert_table

convert_table(*infile*, *outfile=None*, *subtable=None*, *keys=None*, *timecols=None*, *ignorecols=None*, *compressor=None*, *chunks=(10000, - 1)*, *append=False*, *nofile=False*)

Convert casacore table format to xarray Dataset and zarr storage format.

This function requires CASA6 casatools module. Table rows may be renamed or expanded to n-dim arrays based on column values specified in keys.

Parameters

- **infile** (*str*) – Input table filename
- **outfile** (*str*) – Output zarr filename. If None, will use infile name with .tbl.zarr extension
- **subtable** (*str*) – Name of the subtable to process. If None, main table will be used
- **keys** (*dict or str*) – Source column mappings to dimensions. Can be a dict mapping source columns to target dims, use a tuple when combining cols (ie {('ANTENNA1','ANTENNA2'):'baseline'} or a string to rename the row axis dimension to the specified value. Default of None
- **timecols** (*list*) – list of strings specifying column names to convert to datetime format from casacore time. Default is None
- **ignorecols** (*list*) – list of column names to ignore. This is useful if a particular column is causing errors. Default is None
- **compressor** (*numcodecs.blosc.Blosc*) – The blosc compressor to use when saving the converted data to disk using zarr. If None the zstd compression algorithm used with compression level 2.
- **chunks** (*int*) – Shape of desired chunking in the form of (dim0, dim1, . . . , dimN), use -1 for entire axis in one chunk. Default is (80000, 10). Chunking is applied per column / data variable. If too few dimensions are specified, last chunk size is reused as necessary. Note: chunk size is the product of the four numbers, and data is batch processed by the first axis, so that will drive memory needed for conversion.
- **append** (*bool*) – Append an xarray dataset as a new partition to an existing zarr directory. False will overwrite zarr directory with a single new partition
- **nofile** (*bool*) – Allows legacy table to be directly read without file conversion. If set to true, no output file will be written and entire table will be held in memory. Requires ~4x the memory of the table size. Default is False

Returns New xarray Dataset of table data contents. One element in list per DDI plus the metadata global.

Return type New xarray.core.dataset.Dataset

cngi.conversion.describe_ms

describe_ms(*infile*)
Summarize the contents of an MS directory in casacore table format

Parameters **infile** (*str*) – input filename of MS

Returns Summary information

Return type pandas.core.frame.DataFrame

cngi.conversion.read_ms

read_ms(*infile*, *ddis=None*, *ignore=None*, *chunks=(400, 400, 64, 2)*)
Read legacy format MS to xarray Visibility Dataset

The CASA MSv2 format is converted to the MSv3 schema per the specified definition here: https://drive.google.com/file/d/10TZ4dsFw9CconBc-GFxSeb2caT6wkmza/view?usp=sharing

The MS is partitioned by DDI, which guarantees a fixed data shape per partition. This results in separate xarray dataset (xds) partitions contained within a main xds (mxds). There is no DDI in MSv3, so this simply serves as a partition id for each xds.

Parameters

- **infile** (*str*) – Input MS filename
- **ddis** (*list*) – List of specific DDIs to read. DDI's are integer values, or use 'global' string for subtables. Leave as None to read entire MS
- **ignore** (*list*) – List of subtables to ignore (case sensitive and generally all uppercase). This is useful if a particular subtable is causing errors or is very large and slowing down reads. Default is None
- **chunks** (*4-D tuple of ints*) – Shape of desired chunking in the form of (time, baseline, channel, polarization). Larger values reduce the number of chunks and speed up the reads at the cost of more memory. Chunk size is the product of the four numbers. Default is (400, 400, 64, 2)

Returns Main xarray dataset of datasets for this visibility set

Return type xarray.core.dataset.Dataset

cngi.conversion.read_table

read_table(*infile*, *subtable=None*, *timecols=None*, *ignorecols=None*)
Read generic casacore table format to xarray Dataset

Parameters

- **infile** (*str*) – Input table filename
- **subtable** (*str*) – Name of the subtable to process. If None, main table will be used
- **timecols** (*list*) – list of strings specifying column names to convert to datetime format from casacore time. Default is None
- **ignorecols** (*list*) – list of column names to ignore. This is useful if a particular column is causing errors. Default is None

Returns New xarray Dataset of table data contents. One element in list per DDI plus the metadata global.

Return type New xarray.core.dataset.Dataset

convert_image	Convert legacy CASA or FITS format Image to xarray Image Dataset and zarr storage format
convert_ms	Convert legacy format MS to xarray Visibility Dataset and zarr storage format
convert_table	Convert casacore table format to xarray Dataset and zarr storage format.
describe_ms	Summarize the contents of an MS directory in casacore table format
read_ms	Read legacy format MS to xarray Visibility Dataset
read_table	Read generic casacore table format to xarray Dataset

2.1.2 dio

Most CNGI functions operate on xarray Datasets while the data is stored on disk in Zarr format. These functions allow the transition back and forth between the two.

To access these functions, use your favorite variation of: `import cngi.dio`

cngi.dio.append_xds

append_xds(*list_xarray_data_variables*, *outfile*, *chunks_return={}*, *compressor=None*, *graph_name='append_zarr'*)

Append a list of dask arrays to a zarr file on disk. If a data variable with the same name is found it will be overwritten. Data will probably be corrupted if append_zarr overwrites the data variable from which the dask array gets its data. All data variables that share dimensions and coordinates with data variables already on disk must have the same values (chunking can be different).

Parameters

- **list_xarray_data_variables** (*list of dask arrays*) – List of xarray datavariables to append.
- **outfile** (*str*) – The file name of the dataset on disk, generally ends in .zarr
- **chunks_return** (*dict of int*) – A dictionary with the chunk size that will be returned. For example {'time': 20, 'chan': 6}. If chunks_return is not specified the chunking on disk will be used.
- **compressor** (*numcodecs.blosc.Blosc*) – The blosc compressor to use when saving the converted data to disk using zarr. If None the zstd compression algorithm used with compression level 2.
- **graph_name** (*string*) – The time taken to execute the graph and save the dataset is measured and saved as an attribute in the zarr file. The graph_name is the label for this timing information.

cngi.dio.describe_vis

describe_vis(*infile*)
Summarize the contents of a zarr format Visibility directory on disk

Parameters **infile** (*str*) – input filename of zarr Visibility data

Returns Summary information

Return type pandas.core.frame.DataFrame

cngi.dio.read_image

read_image(*infile*, *chunks=None*, *consolidated=True*, *overwrite_encoded_chunks=True*, ***kwargs*)
Read xarray zarr format image from disk

Parameters

- **infile** (*str*) – input zarr image filename
- **chunks** (*dict*) – sets specified chunk size per dimension. Dict is in the form of 'dim':chunk_size, for example {'d0':100, 'd1':100, 'chan':32, 'pol':1}. Default None uses the original zarr chunking.
- **consolidated** (*bool*) – use zarr consolidated metadata capability. Only works for stores that have already been consolidated. Default True works with datasets produced by convert_image which automatically consolidates metadata.
- **overwrite_encoded_chunks** (*bool*) – drop the zarr chunks encoded for each variable when a dataset is loaded with specified chunk sizes. Default True, only applies when chunks is not None.
- **s3_key** (*string, optional*) – optional support for explicit authentication if infile is provided as S3 URL. If S3 url is passed as input but this argument is not specified then only publicly-available, read-only buckets are accessible (so output dataset will be read-only).
- **s3_secret** (*string, optional*) – optional support for explicit authentication if infile is provided as S3 URL. If S3 url is passed as input but this argument is not specified then only publicly-available, read-only buckets are accessible (so output dataset will be read-only).

Returns New xarray Dataset of image contents

Return type xarray.core.dataset.Dataset

cngi.dio.read_vis

read_vis(*infile*, *partition=None*, *chunks=None*, *consolidated=True*, *overwrite_encoded_chunks=True*, ***kwargs*)
Read zarr format Visibility data from disk to xarray Dataset

Parameters

- **infile** (*str*) – input Visibility filename
- **partition** (*string or list*) – name of partition(s) to read as returned by describe_vis. Multiple partitions in list form will return a master dataset of datasets. Use 'global' for global metadata. Default None returns everything

- **chunks** (*dict*) – sets specified chunk size per dimension. Dict is in the form of 'dim':chunk_size, for example {'time':100, 'baseline':400, 'chan':32, 'pol':1}. Default None uses the original zarr chunking.
- **consolidated** (*bool*) – use zarr consolidated metadata capability. Only works for stores that have already been consolidated. Default True works with datasets produced by convert_ms which automatically consolidates metadata.
- **overwrite_encoded_chunks** (*bool*) – drop the zarr chunks encoded for each variable when a dataset is loaded with specified chunk sizes. Default True, only applies when chunks is not None.
- **s3_key** (*string, optional*) – optional support for explicit authentication if infile is provided as S3 URL. If S3 url is passed as input but this argument is not specified then only publicly-available, read-only buckets are accessible (so output dataset will be read-only).
- **s3_secret** (*string, optional*) – optional support for explicit authentication if infile is provided as S3 URL. If S3 url is passed as input but this argument is not specified then only publicly-available, read-only buckets are accessible (so output dataset will be read-only).

Returns New xarray Dataset of Visibility data contents

Return type xarray.core.dataset.Dataset

cngi.dio.write_image

write_image(*xds*, *outfile*, *chunks_return={}*, *chunks_on_disk={}*, *consolidated=True*, *compressor=None*, *graph_name='write_zarr'*)

Write image xarray dataset to zarr format on disk. When chunks_on_disk is not specified the chunking in the input dataset is used. When chunks_on_disk is specified that dataset is saved using that chunking. :param xds: Dataset to write to disk :type xds: xarray.core.dataset.Dataset :param outfile: outfile filename, generally ends in .zarr :type outfile: str :param chunks_on_disk: A dictionary with the chunk size that will be used when writing to disk. For example {'time': 20, 'chan': 6}.

If chunks_on_disk is not specified the chunking of dataset will be used.

Parameters

- **compressor** (*numcodecs.blosc.Blosc*) – The blosc compressor to use when saving the converted data to disk using zarr. If None the zstd compression algorithm used with compression level 2.
- **graph_name** (*string*) – The time taken to execute the graph and save the dataset is measured and saved as an attribute in the zarr file. The graph_name is the label for this timing information.

cngi.dio.write_vis

write_vis(*mxds*, *outfile*, *chunks_on_disk=None*, *partition=None*, *consolidated=True*, *compressor=None*, *graph_name='write_zarr'*)

Write xarray dataset to zarr format on disk. When chunks_on_disk is not specified the chunking in the input dataset is used. When chunks_on_disk is specified that dataset is saved using that chunking.

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – Dataset of dataset to write to disk
- **outfile** (*str*) – outfile filename, generally ends in .zarr
- **chunks_on_disk** (*dict of int*) – A dictionary with the chunk size that will be used when writing to disk. For example {'time': 20, 'chan': 6}. If chunks_on_disk is not specified the chunking of dataset will be used.
- **partition** (*str or list*) – Name of partition xds to write into outfile (from the mxds attributes section). Overwrites existing partition of same name. Default None writes entire mxds
- **compressor** (*numcodecs.blosc.Blosc*) – The blosc compressor to use when saving the converted data to disk using zarr. If None the zstd compression algorithm used with compression level 2.
- **graph_name** (*string*) – The time taken to execute the graph and save the dataset is measured and saved as an attribute in the zarr file. The graph_name is the label for this timing information.

append_xds	Append a list of dask arrays to a zarr file on disk. If a data variable with the same name is found it will be overwritten.
describe_vis	Summarize the contents of a zarr format Visibility directory on disk
read_image	Read xarray zarr format image from disk
read_vis	Read zarr format Visibility data from disk to xarray Dataset
write_image	Write image xarray dataset to zarr format on disk. When chunks_on_disk is not specified the chunking in the input dataset is used.
write_vis	Write xarray dataset to zarr format on disk. When chunks_on_disk is not specified the chunking in the input dataset is used.

2.1.3 direct

These functions allow direct access to the underlying Dask processing engine.

To access these functions, use your favorite variation of: `import cngi.direct`

cngi.direct.framework

InitializeFramework(*workers=2*, *memory='8GB'*, *processes=True*, ***kwargs*)
Initialize the CNGI framework

This sets up the processing environment such that all future calls to Dask Dataframes, arrays, etc will automatically use the scheduler that is configured here.

Parameters

- **workers** (*int*) – Number of processor cores to use, Default=2
- **memory** (*str*) – Max memory allocated to each worker in string format. Default='8GB'
- **processes** (*bool*) – Whether to use processes (True) or threads (False), Default=True
- **threads_per_worker** (*int*) – Only used if processes = True. Number of threads per python worker process, Default=1

Returns Client from Dask Distributed for use by Dask objects

Return type distributed.client.Client

GetFrameworkClient()
Return the CNGI framework scheduler client

Returns Client from Dask Distributed for use by Dask objects

Return type distributed.client.Client

InitializeFramework	Initialize the CNGI framework
GetFrameworkClient	Return the CNGI framework scheduler client

2.1.4 image

These functions examine or manipulate Image data in the xarray Dataset (xds) format. They take an xds as input and return a new xds or some other structure as output. Some may operate directly on the zarr data store on disk.

The input xarray Dataset is never modified.

To access these functions, use your favorite variation of: `import cngi.image`

cngi.image.cont_sub

cont_sub(*xds*, *dv='IMAGE'*, *fitorder=2*, *chans=None*, *linename='LINE'*, *contname='CONTINUUM'*, *compute=False*)
Continuum subtraction of an image cube

Perform a polynomial baseline fit to the specified channels from an image and subtract it from all channels

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – image xarray dataset
- **dv** (*str*) – name of data_var in xds to polynomal fit. Default is 'IMAGE'
- **chans** (*array of int*) – Spectral channels to use for fitting a polynomial to determine continuum. Default is None, use all channels.
- **fitorder** (*int*) – Order of polynomial to fit to the specified spectral channels to determine the continuum. Default is 2.

- **linename** (*str*) – dataset variable name for output: name of image to which to save the result of subtracting the computed continuum from the input image. overwrites if already present. Default is 'LINE'
- **contname** (*str*) – dataset variable name for outputthe computed continuum image. overwrites if already present. Default is 'CONTINUUM'
- **compute** (*bool*) – execute the DAG to compute the fit error. Default False returns lazy DAG (error can then be retrieved via xds.<name>.<key>.values)

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.fit_gaussian

fit_gaussian(*xds*, *dv='PSF'*, *beam_set_name='RESTORE_PARMS'*, *npix_window=[9, 9]*, *sampling=[9, 9]*, *cutoff=0.35*)

Fit one or more elliptical gaussian components on an image region

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image xarray dataset
- **dv** (*str*) – image data variable to fit. Default is 'PSF'

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.fit_gaussian_rl

fit_gaussian_rl(*xds*, *dv='PSF'*, *beam_set_name='RESTORE_PARMS'*, *fit_method='rm_fit'*, *npix_window=[21, 21]*, *sampling=[401, 401]*, *cutoff=0.5*, *cutoff_sensitivity=0.003*)

Fit one or more elliptical gaussian components on an image region

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image xarray dataset
- **dv** (*str*) – image data variable to fit. Default is 'PSF'

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.gaussian_beam

gaussian_beam(*xds*, *source='commonbeam'*, *scale=1.0*, *name='BEAM'*)

Construct a gaussian beam of image dimensions from specified size or beam attribute

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **source** (*str or list of floats*) – Source xds attr name to find the beam information, or a list describing the major axis, minor axis, and position angle of the desired gaussian in (arcsec, arcsec, degrees), for example [1., 1., 30.]. Default is 'commonbeam'
- **scale** (*float*) – peak amplitude of beam. Default is unity (1.0)

- **name** (*str*) – dataset variable name for output beam(s), overwrites if already present. Default is 'BEAM'

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.implot

implot(*xda*, *axis=['l', 'm']*, *chans=None*, *pols=None*, *overplot=False*, *drawplot=True*, *tsize=250*, *title=None*)

Plot a preview of Image xarray DataArray contents

Parameters

- **xda** (*xarray.core.dataarray.DataArray*) – input DataArray
- **axis** (*str or list*) – DataArray coordinate(s) to plot against data. Default ['d0', 'd1']. All other coordinates will be averaged
- **chans** (*int or list of ints*) – channel axis indices to select prior to averaging
- **pols** (*int or list of ints*) – polarization axis indices to select prior to averaging
- **overplot** (*bool*) – Overlay new plot on to existing window. Default of False makes a new window for each plot
- **drawplot** (*bool*) – Display plot window. Should pretty much always be True unless you want to overlay things in a Jupyter notebook.
- **tsize** (*int*) – target size of the preview plot (might be smaller). Default is 250 points per axis

Returns

Return type Open matplotlib window

cngi.image.mask

mask(*xds*, *name='MASK1'*, *ra=None*, *dec=None*, *pixels=None*, *pol=- 1*, *channels=- 1*)

Create a new mask Data variable in the Dataset

Note: This function currently only supports rectangles and integer pixel boundaries

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **name** (*str*) – dataset variable name for mask, overwrites if already present
- **ra** (*list*) – right ascension coordinate range in the form of [min, max]. Default None means all
- **dec** (*list*) – declination coordinate range in the form of [min, max]. Default None means all
- **pixels** (*numpy.ndarray*) – array of shape (N,2) containing pixel box. AND'd with ra/dec

- **pol** (*int or list*) – polarization dimension(s) to include in mask. Default of -1 means all
- **channels** (*int or list*) – channel dimension(s) to include in mask. Default of -1 means all

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.moments

moments (*xds*, *dv='IMAGE'*, *moment=None*, *axis='chan'*)

Collapse an n-dimensional image cube into a moment by taking a linear combination of individual planes

---

Note: This implementation still needs to implement additional moment codes, and verify behavior of implemented moment codes.

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **dv** (*str*) – image data variable. Default is 'IMAGE'
- **moment** (*int or list of ints*) – number that selects which moment to calculate from the following list -1 - mean value of the spectrum (default) 0 - integrated value of the spectrum 1 - intensity weighted coordinate; traditionally used to get 'velocity fields' 2 - intensity weighted dispersion of the coordinate; traditionally used to get 'velocity dispersion' 3 - median of I 4 - median coordinate 5 - standard deviation about the mean of the spectrum 6 - root mean square of the spectrum 7 - absolute mean deviation of the spectrum 8 - maximum value of the spectrum 9 - coordinate of the maximum value of the spectrum 10 - minimum value of the spectrum 11 - coordinate of the minimum value of the spectrum
- **axis** (*str*) – specified axis along which to reduce for moment generation, Default='chan'

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.rebin

rebin (*xds*, *factor=1*, *axis='chan'*)

Rebin an n-dimensional image across any single (spatial or spectral) axis

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **factor** (*int*) – scaling factor for binning, Default=1 (no change)
- **axis** (*str*) – dataset dimension upon which to rebin ('d0', 'd1', 'chan', 'pol'). Default is 'chan'

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.reframe

reframe(*ixds*, *outframe=None*, *reference_time=None*, *observer_location=None*, *target_location=None*, *reference_frequency=None*)

Change the velocity system of an image

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image
- **outframe** (*str*) – reference frame to which the input will be converted
- **reference_time** (*numpy.datetime64 or datetime, optional*) – any valid initializer of astropy.time.TimeBase should work
- **observer_location** (*astropy.cordinates.SkyCoord, optional*) – position and velocity of observer for frame transformation
- **target_location** (*astropy.cordinates.SkyCoord, optional*) – position and velocity of source for frame transformation
- **reference_frequency** (*astropy.units.Quantity, optional*) – value and unit to use when expressing the spectral value as a velocity, input to SpectralCoord doppler_rest parameter

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.region

region(*xds*, *name='REGION1'*, *ra=None*, *dec=None*, *pixels=None*, *pol=- 1*, *channels=- 1*)

Create a new region Data variable in the Dataset

Note: This function currently only supports rectangles and integer pixel boundaries

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input image dataset
- **name** (*str*) – dataset variable name for region, overwrites if already present
- **ra** (*list*) – right ascension coordinate range in the form of [min, max]. Default None means all
- **dec** (*list*) – declination coordinate range in the form of [min, max]. Default None means all
- **pixels** (*array_like*) – array of shape (N,2) containing pixel box. OR'd with ra/dec
- **pol** (*int or list*) – polarization dimension(s) to include in region. Default of -1 means all
- **channels** (*int or list*) – channel dimension(s) to include in region. Default of -1 means all

Returns New Dataset

Return type xarray.core.dataset.Dataset

cngi.image.smooth

smooth(*xds*, *dv='IMAGE'*, *kernel='gaussian'*, *size=[1.0, 1.0, 30.0]*, *current=None*, *scale=1.0*, *name='BEAM'*)

Smooth data along the spatial plane of the image cube.

Computes a correcting beam to produce defined size when kernel=gaussian and current is defined. Otherwise the size or existing beam is used directly.

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **dv** (*str*) – name of data_var in xds to smooth. Default is 'IMAGE'
- **kernel** (*str*) – Type of kernel to use:'boxcar', 'gaussian' or the name of a data var in this xds. Default is 'gaussian'.
- **size** (*list of floats*) – for gaussian kernel, list of three values corresponding to major and minor axes (in arcseconds) and position angle (in degrees). for boxcar kernel, list of two valuess corresponding to l,m bin width. Default is [1., 1., 30.] (for a gaussian)
- **current** (*list of floats*) – same structure as size, a list of three values corresponding to major and minor axes (in arcseconds) and position angle (in degrees) of the current beam applied to the image. Default is None
- **scale** (*float*) – gain factor after convolution. Default is unity gain (1.0)
- **name** (*str*) – dataset variable name for kernel, overwrites if already present

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.spec_fit

spec_fit(*xds*, *dv='IMAGE'*, *pixel=(0.5, 0.5)*, *pol=0*, *sigma=2000*, *name='FIT'*)

Perform gaussian spectral line fits in the image cube

Adapted from https://github.com/emilyripka/BlogRepo/blob/master/181119_PeakFitting.ipynb Dave Mehringer 2021mar01

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **dv** (*str*) – name of data_var in xds to smooth. Default is 'IMAGE'
- **pixel** (*tuple of int or float*) – tuple of integer or float coordinates of pixel to fit. If int, pixel index is used. If float, nearest pixel at that fractional location is used. Default is (0.5,0.5) corresponding to center pixel
- **pol** (*int*) – polarization index to use. Default is 0
- **sigma** (*float*) – sigma of gaussian fit. Default is 1000
- **name** (*str*) – dataset variable name fit, overwrites if already present

Returns output Image with name added fit results in attributes

Return type xarray.core.dataset.Dataset

cngi.image.statistics

statistics(*xds*, *dv='IMAGE'*, *name='statistics'*, *compute=False*)
Generate statistics on specified image data contents

Resulting data is placed in the attributes section of the dataset

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – input Image Dataset
- **dv** (*str*) – name of data_var in xds to compute statistics on. Default is 'IMAGE'
- **name** (*str*) – name of the attribute in xds to hold statistics dictionary. Default is 'statistics'
- **compute** (*bool*) – execute the DAG to compute the statistics. Default False returns lazy DAG (statistics can then be retrieved via xds.<name>.<key>.values)

Returns output Image

Return type xarray.core.dataset.Dataset

cngi.image.stokes_to_corr

stokes_to_corr(*ixds*, *basis='linear'*)
Convert polarization data from Stokes parameters to the correlation basis.

To be used as a converter during image reconstruction, and also stand-alone.

The sign convention used here should be the same as what CASA uses, (XX^YY)/2

..todo: Apply transformations to all data variables, not just IMAGE

..todo: Drop the basis and code_name dimensions from the product

..todo: Update pol dimension to match reassigned version across the rest of the xds when combining

..todo: Ensure that math to calculate linear basis products matches the CASA convention

Parameters

- **ixds** (*xarray.core.dataset.Dataset*) – Input image dataset (e.g., loaded from img.zarr file) with polarization data as Stokes (I,Q,U,V) parameters.
- **basis** (*string, default='linear'*) – Desired correlation basis 'linear' or 'circular'

Returns Output image dataset with polarization data in the linear (XX,XY,YX,YY) or circular (RR,RL,LR,LL) correlation basis.

Return type xarray.core.dataset.Dataset

See also:

`corr_to_stokes`

Notes

Polarization codes from the MeasurementSet are preserved in vis.zarr: #. I #. Q #. U #. V #. RR #. RL #. LR #. LL #. XX #. XY #. YX #. YY

Raises

- **`UserWarning`** – If input pol dimension does not contain expected codes, or has the wrong shape.
- **`NotImplementedError`** – If the input image has less than 4 Stokes parameters we still compute the possible results but cannot return them, so trap a ValueError with this until it is decided how to align pol dimension in the converted image with the pol dimension for all the other data variables of the input. Is it necessary to update the codes along the pol dimension for the whole dataset to match new basis?
- **`. note::`** – Support is presently limited for heterogeneous-feed arrays with elements expected to be missing in a given basis (e.g., very long baseline interferometry with the Event Horizon Telescope).

cont_sub	Continuum subtraction of an image cube
fit_gaussian	Fit one or more elliptical gaussian components on an image region
fit_gaussian_rl	Fit one or more elliptical gaussian components on an image region
gaussian_beam	Construct a gaussian beam of image dimensions from specified size or beam attribute
implot	Plot a preview of Image xarray DataArray contents
mask	Create a new mask Data variable in the Dataset
moments	Collapse an n-dimensional image cube into a moment by taking a linear combination of individual planes
rebin	Rebin an n-dimensional image across any single (spatial or spectral) axis
reframe	Change the velocity system of an image
region	Create a new region Data variable in the Dataset
smooth	Smooth data along the spatial plane of the image cube.
spec_fit	Perform gaussian spectral line fits in the image cube
statistics	Generate statistics on specified image data contents
stokes_to_corr	Convert polarization data from Stokes parameters to the correlation basis.

2.1.5 vis

These functions examine or manipulate Visibility data in the xarray Dataset (xds) format. They take an xds as input and return a new xds or some other structure as output. Some may operate directly on the zarr data store on disk.

The input xarray Dataset is never modified.

To access these functions, use your favorite variation of: `import cngi.vis`

cngi.vis.apply_flags

apply_flags (*mxds*, *vis*, *flags='FLAG'*)
Apply flag variables to other data in Visibility Dataset

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis** (*str*) – visibility partition in the mxds to use
- **flags** (*list or str*) – data var name or list of names to use as flags. Default 'FLAG' uses the FLAG field

Returns output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

cngi.vis.chan_average

chan_average (*mxds*, *vis*, *width=1*)
Average data across the channel axis

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis** (*str*) – visibility partition in the mxds to use
- **width** (*int*) – number of adjacent channels to average. Default=1 (no change)

Returns New output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

cngi.vis.chan_smooth

chan_smooth (*mxds*, *vis*, *type='triang'*, *size=3*, *gain=1.0*, *window=None*)
Apply a smoothing kernel to the channel axis

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis** (*str*) – visibility partition in the mxds to use
- **type** (*str or tuple*) – type of window function to use: 'boxcar', 'triang', 'hann' etc. Default is 'triang'. Scipy.signal is used to generate the window weights, refer to https://docs.scipy.org/doc/scipy/reference/signal.windows.html#module-scipy.signal.windows for a complete list of supported windows. If your window choice requires additional parameters, use a tuple e.g. ('exponential', None, 0.6)
- **size** (*int*) – width of window (# of channels). Default is 3
- **gain** (*float*) – gain factor after convolution. Used to set weights. Default is unity gain (1.0)
- **window** (*list of floats*) – user defined window weights to apply (all other options ignored if this is supplied). Default is None

Returns New output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

cngi.vis.join_dataset

join_dataset(*mxds1*, *mxds2*)

Join together two visibility zarr directories.

Creates a new mxds with all the visibilities "xds*" of mxds1/mxds2 and all the subtables of mxds1 and mxds2. Visibilities are renamed so as not to collide where necessary. Subtable values are preserved where they are equal, and updated to have new dimensional coordinate values where they are not equal.

The order of the visibilities in the two mxds is preserved. If mxds1 and mxds2 have visibilits ["xds0", "xds2"] and ["xds1", "xds2"], respectively, then the new mxds will have visibilities ["xds0", "xds2", "xds3", "xds4"], where the visibilities from mxds2 got renamed xds1->xds3 and xds2->xds4.

Parameters

- **mxds1** (*xarray.core.dataset.Dataset*) – First multi-xarray Dataset with global data to join.
- **mxds2** (*xarray.core.dataset.Dataset*) – Second multi-xarray Dataset with global data to join.

Returns New output multi-xarray Dataset with global data.

Return type xarray.core.dataset.Dataset

cngi.vis.join_vis

join_vis(*mxds*, *vis1*, *vis2*)

Concatenate together two Visibility xds's of compatible shape from the same mxds

The data variables of the two datasets are merged together, with some limitations (see "Current Limitations" in the Notes section).

Coordinate values that are not also being used as dimensions are compared for equality.

Certain known attributes are updated, namely "ddi". For the rest, they are merged where keys are present in one dataset but not the other, or the values from the first dataset override those from the second where the keys are the same.

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis1** (*str*) – first visibility partition in the mxds to join
- **vis2** (*str*) – second visibility partition in the mxds to join

Returns New output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

Warning: Joins are highly discouraged for datasets that don't share a common 'global' DDI (ie are sourced from different .zarr archives). Think really hard about if a join would even mean anything before doing so.

Warning: DDIs are separated by spectral window and correlation (polarity) because it is a good indicator of how data is gathered in hardware. Ultimately, if source data comes from different DDIs, it means that the data followed different paths through hardware during the measurement. This is important in that there are more likely to be discontinuities across DDIs than within them. Therefore, don't haphazardly join DDIs, because it could break the inherent link between data and hardware.

Notes

Conflicts in data variable values between datasets:

There are many ways that data values could end up differing between datasets for the same coordinate values. One example is the error bars represented in SIGMA or WEIGHT could differ for a different spw.

There are many possible solutions to dealing with conflicting data values:

1. only allow joins that don't have conflicts (current solution)
2. add extra indexes CHAN, POL, and/or SPW to the data variables that conflict
3. add extra indexes CHAN, POL, and/or SPW to all data variables
4. numerically merge the values (average, max, min, etc)
5. override the values in xds1 with the values in xds2

Joins are allowed for:

Datasets that have all different dimension values.

- Example: xds1 covers time range 22:00-22:59, and xds2 covers time range 23:00-24:00

Datasets that have overlapping dimension values with matching data values at all of those coordinates.

- Example: xds1.PROCESSOR_ID[0][0] == xds2.PROCESSOR_ID[0][0]

Current Limitations:

Joins are not allowed for datasets that have overlapping dimension values with mismatched data values at any of those coordinates.

- Example: xds1.PROCESSOR_ID[0][0] != xds2.PROCESSOR_ID[0][0]
- See "Conflicts in data variable values", above

Joins between 'global' datasets, such as those returned by cngi.dio.read_vis(ddi='global'), are probably meaningless and should be avoided. Datasets do not need to have the same shape.

- Example xds1.DATA.shape != xds2.DATA.shape

Examples

Use cases (some of them), to be turned into examples. Note: these use cases come from CASA's mstransform(combinespws=True) and may not apply to ddijoin.

- universal calibration across spws
- autoflagging with broadband rfi
- uvcontfit and uvcontsub
- joining datasets that had previously been split, operated on, and are now being re-joined

cngi.vis.reframe

reframe(*mxds*, *vis*, *mode='channel'*, *nchan=None*, *start=0*, *width=1*, *interpolation='linear'*, *phasecenter=None*, *restfreq=None*, *outframe=None*, *veltype='radio'*)

Transform channel labels and visibilities to a spectral reference frame which is appropriate for analysis, e.g. from TOPO to LSRK or to correct for doppler shifts throughout the time of observation

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis** (*str*) – visibility partition in the mxds to use
- **nchan** (*int*) – number of channels in output spw. None=all
- **start** (*int*) – first input channel to use
- **width** (*int*) – number of input channels to average
- **interpolation** (*str*) – spectral interpolation method
- **phasecenter** (*int*) – image phase center position or field index
- **restfreq** (*float*) – rest frequency
- **outframe** (*str*) – output frame, None=keep input frame
- **veltype** (*str*) – velocity definition

Returns New output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

cngi.vis.split_dataset

split_dataset(*mxds: xarray.Dataset*, *xds_names: Union[str, List[str]]*) → xarray.Dataset

Pull the xds visibilites out with the mxds, preserving only that information in the subtables that is related to the given visibilities.

Creates a new mxds to return based off the input mxds. Only the visibilites mentioned in xds_names are included. Subtable data is reduced to only include related information, based on the relational keys in the visibility tables. Finally, the coordinate values of the new mxds are updated to reflect the limited coordinate values in the included visibilities.

Parameters

- **mxds** (*xarray.Dataset*) – The multi-xds dataset to pull data out of.
- **xds_names** (*str or list*) – Name(s) of the visibilities dataset. Each name should be of the form "xds*"

Returns A new mxds, which includes just the xds_names visibility Dataset(s) and the related information from the mxds subtables.

Return type xarray.Dataset

cngi.vis.time_average

time_average(*mxds*, *vis*, *bin=1*, *width=None*, *span='state'*, *maxuvwdistance=None*)
Average data across the time axis

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis** (*str*) – visibility partition in the mxds to use
- **bin** (*int*) – number of adjacent times to average, used when width is None. Default=1 (no change)
- **width** (*str*) – resample to width freq (i.e. '10s') and produce uniform time steps over span. Ignores bin. Default None uses bin value. see https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html.
- **span** (*str*) – span of the binning. Allowed values are 'scan', 'state' or 'both'. Default is 'state' (meaning all states in a scan)
- **(future)** (*maxuvwdistance*) – NOT IMPLEMENTED. maximum separation of start-to-end baselines that can be included in an average. (meters)

Returns New output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

cngi.vis.uv_cont_fit

uv_cont_fit(*mxds*, *vis*, *source='DATA'*, *target='CONTFIT'*, *fitorder=1*, *excludechans=[]*)
Fit a polynomial regression to source data and return model values to target

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – input multi-xarray Dataset with global data
- **vis** (*str*) – visibility partition in the mxds to use
- **source** (*str*) – data variable in the dataset on which to fit the regression. Default is 'DATA'
- **target** (*str*) – new data variable to place the fit result, overwrites if already present. Default is 'CONTFIT'
- **fitorder** (*int*) – polynomial order for the fit, must be >= 1, but values larger than 1 will slow down rapidly. Default is 1
- **excludechans** (*list of ints*) – indices of channels to exclude from the fit. Default is empty (include all channels)

Returns New output multi-xarray Dataset with global data

Return type xarray.core.dataset.Dataset

cngi.vis.visplot

visplot (*xda*, *axis=None*, *overplot=False*, *drawplot=True*, *tsize=250*)
Plot a preview of Visibility xarray DataArray contents

Parameters

- **xda** (*xarray.core.dataarray.DataArray*) – input DataArray to plot
- **axis** (*str or list or xarray.core.dataarray.DataArray*) – Coordinate(s) within the xarray DataArray, or a second xarray DataArray to plot against. Default None uses range. All other coordinates will be maxed across dims
- **overplot** (*bool*) – Overlay new plot on to existing window. Default of False makes a new window for each plot
- **drawplot** (*bool*) – Display plot window. Should pretty much always be True unless you want to overlay things in a Jupyter notebook.
- **tsize** (*int*) – target size of the preview plot (might be smaller). Default is 250 points per axis

Returns

Return type Open matplotlib window

apply_flags	Apply flag variables to other data in Visibility Dataset
chan_average	Average data across the channel axis
chan_smooth	Apply a smoothing kernel to the channel axis
join_dataset	Join together two visibility zarr directories.
join_vis	Concatenate together two Visibility xds's of compatible shape from the same mxds
reframe	Transform channel labels and visibilities to a spectral reference frame which is appropriate for analysis, e.g. from TOPO to LSRK or to correct for doppler shifts throughout the time of observation
split_dataset	Pull the xds visibilites out with the mxds, preserving only that information in the subtables that is related to the given visibilities.
time_average	Average data across the time axis
uv_cont_fit	Fit a polynomial regression to source data and return model values to target
visplot	Plot a preview of Visibility xarray DataArray contents

2.2 ngcasa

ngCASA Package

Module functions with blank descriptions have not been completed yet. As each function is completed, its description will be populated.

2.2.1 calibration

Calibration subpackage modules

ngcasa.calibration.apply_calibration

apply_calibration(*vis_dataset*, *cal_dataset*, *apply_parms*, *storage_parms*)

Todo: This function is not yet implemented

Apply antenna gain solutions according to the parameters in solpars.

Calculate V_ij(corrected) = V_ij(observed) / g_i g_j*

Inputs :

List of calibration solution datasets (to apply in the specified order) Interpolation type ? Data array on which to operate. Default='DATA' Data array in which to fill the output. Default='CORRECTED_DATA' - this option exists in order to support simulator's corrupt operation where we'd pick 'DATA'

TBD : Should this translation of the caltable back to the original un-averaged data be done here, or in a centralized CNGI method that handles such interpolations for all methods that need to convert averaged values into un-averaged values. Note the difference between just copying and expanding values, versus interpolation.

TBD : Single apply() method, or several ?

Returns **vis_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.calibration.self_cal

self_cal(*vis_mxds*, *solve_parms*, *sel_parms*)

Todo: This function is not yet implemented

Calculate antenna gain solutions according to the parameters in solpars. The input dataset has been pre-averaged/processed and the model visibilities exist

Iteratively solve the system of equations g_i g_j* = V_data_ij/V_model_ij for all ij. Construct a separate solution for each timestep and channel in the input dataset.

Options :

amp, phase or both solution type (?) G-term, D-term, etc... Data array for which to calculate solutions. Default='DATA'

TBD :

Single method with options for solutions of different types ? Or, separate methods for G/B, D, P etc.. : solve_B, solve_D, solve_B, etc. . .

apply_calibration

self_cal

2.2.2 deconvolution

Deconvolution subpackage modules

ngcasa.deconvolution.deconvolve_adaptive_scale_pixel

deconvolve_adaptive_scale_pixel(*img_dataset*, *deconvolve_parms*, *storage_parms*)

Todo: This function is not yet implemented

An iterative solver to construct a 2D mixed model from an observed image(set) and psf(set).

Sky Model - A linear combination of 2D Gaussians Algorithm - Chi-square / TV minimization on atom parameters, with subspace selections.

Options - Narrow-band, Wide-band

Input - Requires an input cube (mfs is a cube with nchan=1) Output - Cube model image and/or a list of flux components.

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.deconvolve_fast_resolve

deconvolve_fast_resolve(*img_dataset*, *deconvolve_parms*, *storage_parms*)

Todo: This function is not yet implemented

An iterative solver to construct a Bayesian model from an observed image(set) and psf(set).

Sky Model - Pixel amplitudes Algorithm - Bayesian formulation that includes constraints on the flux distribution and wideband support.

Input - Requires an input cube (mfs is a cube with nchan=1) Output - Cube model image, Error map (Spectral index map)

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.deconvolve_multiterm_clean

deconvolve_multiterm_clean(*img_dataset*, *deconvolve_parms*, *storage_parms*)

Todo: This function is not yet implemented

An iterative solver to construct a model from an observed image(set) and psf(set).

Sky model - A (multi-term) linear combination of basis functions.

Multi-scale : Basis functions are inverted tapered paraboloids

Multi-scale MFS : Basis functions are Taylor polynomials in frequency

Options :

- **MS-Clean - Multi-scale CLEAN (MS-MFS Clean with nterms=1)**
  - Input - Requires an input cube (mfs is a cube with nchan=1)
  - Output - Cube model image
- **MS-MFS Clean - Wideband Imaging that solves for a set of Taylor coefficient maps.**
  - Input - Multi-channel cube.
  - Output : Taylor coefficient maps, Spectral Index + Evaluation of the model to a Cube model image

Step (1) cngi.image.cube_to_mfs()

Step (2) Implement the multi-term deconvolution algorithm

Step (3) cngi.image.mfs_to_cube()

The special case of nscales=1 and nterms=1 is the same use-case as deconvolve_point_clean.

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.deconvolve_point_clean

deconvolve_point_clean(*img_xds*, *deconvolve_parms*, *sel_parms*)

Todo: This function is not yet implemented

An iterative solver to construct a model from an observed image(set) and psf(set).

Sky model : Point source

Algorithm : CLEAN (a greedy algorithm for chi-square minimization)

Options : Hogbom, Clark

Input : Requires an input cube (mfs is a cube with nchan=1)

Output : Cube model image

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.deconvolve_rotation_measure_clean

deconvolve_rotation_measure_clean(*img_dataset*, *deconvolve_parms*, *storage_parms*)

Todo: This function is not yet implemented

An iterative solver to construct a full-polarization model from an observed image(set) and psf(set).

Sky Model : Per flux component, delta-functions in lambda-square space

Algorithm :

- Step (1) : Transform the cube to lambda-square space
- Step (2) : Construct a 3D RM-synthesis PSF
- Step (3) : Run CLEAN based-deconvolution
- Step (4) : Transform back to frequency space.

Input : Requires an input cube (mfs is a cube with nchan=1)

Output : Cube model image, Error map (Spectral index map)

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.feather

feather(*img_dataset_lowres*, *img_dataset_highres*)

Todo: This function is not yet implemented

Feather two images together, based on restoring beam information stored in both.

Output image = iFT(FT(lowres_image) + [1-FT(lowres_beam)] x FT(highres_image))

TBD : Do this for the entire image_set (psf, image) and updated restoring-beam information as well ?

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.is_converged

is_converged(*img_dataset*, *iterpars*, *storage_parms*)

Todo: This function is not yet implemented

An iteration controller for image reconstruction

The current image set (residual, psf, model, etc) is evaluated against stopping criteria derived from input parameters (iterpars) and the image set itself.

Step 1 : Derive stopping criteria

- Merge explicit user-parameters in iterpars (niter,threshold,etc..) with criteria that are calculated from the imageset (psfsidelobelevel, cyclethreshold, N-sigma-based thresholds, mask-sensitive thresholds)
- Calculate 'cycleniter' and 'cyclethreshold' to be used in Step 2.

Step 2 : Apply stopping criteria (as an ordered list)

- Peak residual within the mask region for imagename.residual <= threshold
- Total iters done >= niter

Returns **img_dataset** – An convergence history list of dict is added to the attributes of img_dataset.

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.linear_mosaic

linear_mosaic(*img_dataset*, *img_mosaic*)

Todo: This function is not yet implemented

Construct a linear mosaic as a primary-beam weighted sum of a set of input images. Individual images are re-sampled onto a larger image grid and summed.

Assume flat-noise normalization for the inputs. (TBD : Or flatsky?)

Output image : sum(input_images) / sum (input_pbs)

TBD :

This requires some sort of merging of img_datasets. CNGI demo on how to append/add images to an image_set and ensure that meta-data are consistent?

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.make_mask

make_mask(*img_dataset*, *mask_parms*, *storage_parms*)

Todo: This function is not yet implemented

Make a region to identify a mask for use in deconvolution.

One or more of the following options are allowed

- Supply a mask in the form of a cngi.image.region
- Run an auto-masking algorithm to detect structure and define a cngi.image.region
- Apply a pblimit based mask

An existing deconvolution mask from img_dataset may either be included in the above, or ignored.

The output is a region (array?) in the img_dataset containing the intersection of all above regions

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.deconvolution.restore_model

restore_model(*img_dataset*, *restore_parms*)

Todo: This function is not yet implemented

Restore a deconvolved model.

Inputs - target resolution could be native or 'common' or explicitly specified.

Cube and single-term imaging :

- Smooth the model image (Jy/pixel) to the target resolution
- Smooth the residual image (Jy/beam) to the target resolution
- Add the two smoothed images

Multi-term imaging :

- Smooth the model taylor coefficient images to the target resolution
- Apply the inverse Hessian to the residual image vector (data-space to model-space) (At non-native target resolution, also compute a new Hessian matched to the scale of the restoring beam.)
- Smooth the model-space residuals to the target resolution

Re-restoration may be done simply by calling this same method again with a different target resolution. Calculations will start with the native model and residual images. Note that re-restoration with cngi.image.imsmooth() will not be accurate for multi-term imaging.

Returns **vis_dataset**

Return type xarray.core.dataset.Dataset

deconvolve_adaptive_scale_pixel
deconvolve_fast_resolve
deconvolve_multiterm_clean
deconvolve_point_clean
deconvolve_rotation_measure_clean
feather
is_converged
linear_mosaic
make_mask
restore_model

2.2.3 flagging

Flagging subpackage modules

These functions can be used to calculate flags and handle different flag versions. There are functions to inspect flag variables and manage the set of flag variables in a CNGI Dataset. Other functions calculate flags using different flagging methods.

ngcasa.flagging.auto_clip

auto_clip (*vis_dataset*, *clip_min*, *clip_max*)

Apply the clip flagging method. Data with values lower than clip_min or bigger than clip_max are flagged. Values are compared against the abs of the visibility values (no other expression supported at the moment).

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.
- **clip_min** (*float*) – Minimum below which data should be flagged
- **max_clip** (*float*) – Maximum above which data should be flagged
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated flags

ngcasa.flagging.auto_rflag

auto_rflag(*vis_dataset*, ***kwargs*)

Todo: This function is not yet implemented

An autoflag algorithm that detects outliers via hierarchical MAD statistics applied to the visibility data.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.
- **TBD** –
- **Returns** –
- **-------** –
- **xarray.core.dataset.Dataset** – Visibility dataset with updated flags

ngcasa.flagging.auto_tfcrop

auto_tfcrop(*vis_dataset*, ***kwargs*)

Todo: This function is not yet implemented

An autoflag algorithm that detects outliers based on the assumption that the time-frequency plane of the visibilities for a sky signal is smooth in comparison to RFI.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.
- **TBD** –
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated flags

ngcasa.flagging.auto_uvbin

auto_uvbin(*vis_dataset*, ***kwargs*)

Todo: This function is not yet implemented

An autoflag algorithm that detects outliers on the gridded spatial frequency plane (Algorithm prototype exists).

TBD: How can this method call ngcasa.imaging._make_grid() and also satisfy code structure rules?

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.

- **TBD** –
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated flags

ngcasa.flagging.elevation

elevation(*vis_dataset*, ***kwargs*)

Todo: This function is not yet implemented

Flag data for low elevations

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.
- **- tolerance** (*TBD*) –
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated flags

ngcasa.flagging.extend

extend(*xds*, ***kwargs*)

Todo: This function is not yet implemented

Parameters

- **xds** (*xarray.core.dataset.Dataset*) – Input dataset.
- **- additional parameters and functions for different methods** (*TBD*) – grow-around, extendflags, growtime, growfreq, antint, etc.

Returns Visibility dataset with updated flags

Return type xarray.core.dataset.Dataset

ngcasa.flagging.manager_add

manager_add(*vis_dataset*, *name*, *descr*, *source=None*)

Add a new flag variable to the dataset. All flags in the new variable take the values from the source flag variable. If no source is found, the flags are all set to false.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset
- **name** (*string*) – The new flag variable name
- **descr** (*string*) – Text description of the flag variable (for example, 'backup_beginning')
- **source** (*name*) – Name of an existing flag variable. If specified its values will be used to initialize the new flag variable being added
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated set of flag variables

ngcasa.flagging.manager_list

manager_list(*vis_dataset*)

Add a new flag variable to the dataset. All flags in the new variable are set to false, unless a source variable is given in which case the values of the source are copied.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset
- **Returns** –
- **-------** –
- **pandas.core.frame.DataFrame** – Information on flag variables from the input dataset

ngcasa.flagging.manager_remove

manager_remove(*vis_dataset*, *name*)

Remove flag variable from the dataset.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset
- **name** (*string*) – The flag variable name to remove
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset without the removed flag variable

ngcasa.flagging.manual_flag

manual_flag(*mxds*, *xds_idx*, *commands=None*, *cmd_filename=None*)

Implements the 'manual' flagging method (equivalent to CASA6's flagdata manual mode).

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – Input Dataset
- **xds_idx** (*int*) – Index of the xarray datasets to get counts from (index in the xds'i' attributes of mxds). This is an oversimplification (early prototyping)
- **commands** (*List[Dict]*) – List of selection commands. Each item in the list represent one selection of data to flag. Every selection item is a dictionary. Selection is currently supported by 'time', 'chan', 'antenna', and 'pol'. 'time', 'chan', and 'pol' correspond directly to the dimensions of FLAG and DATA varaibles in the xarray datasets. 'antenna' is translated to the 'baseline' dimension.
- **cmd_filename** (*filename*) – Name of a file with text flagging commands, using the same format as used in the CASA6 pipelines for the "*flagonline.txt" or "*flagcmds.txt" files.
- **- Additional selection parameters / criteria** (*TBD*) –
- **Returns** –
- **-------** –
- **xarray.core.dataset.Dataset** – Dataset with flags set on the selections given in commands and/or cmd_filename

ngcasa.flagging.manual_unflag

manual_unflag(*mxds*, *xds_idx*, *commands=None*)

Unflags the selected data. Flags corresponding to the selections are unset.

Parameters

- **mxds** (*xarray.core.dataset.Dataset*) – Input Dataset
- **xds_idx** (*int*) – Index of the xarray datasets to get counts from (index in the xds'i' attributes of mxds). This is an oversimplification (early prototyping)
- **commands** (*List[Dict]*) – List of selections, each expressed as an xarray selection dictionary, using the same schema as in manual_flag. If empty, unflag all.
- **- Additional selection parameters / criteria** (*TBD*) –
- **Returns** –
- **-------** –
- **xarray.core.dataset.Dataset** – Visibility dataset with updated (unset) flags

ngcasa.flagging.quack

quack(*vis_dataset*, ***kwargs*)

Todo: This function is not yet implemented

Flag the beginning and/or end of scans to account for observation effects such as antenna slewing delays.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.
- **- time-width** (*TBD*) –
- **or end or both** (*beginning*) –
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated flags

ngcasa.flagging.shadow

shadow(*vis_dataset*, ***kwargs*)

Todo: This function is not yet implemented

Flag all baselines for antennas that are shadowed beyond the specified tolerance.

All antennas in the zarr-file metadata (and their corresponding diameters) will be considered for shadow-flag calculations. For a given timestep, an antenna is flagged if any of its baselines (projected onto the uv-plane) is shorter than

radius_1 + radius_2 - tolerance.

The value of 'w' is used to determine which antenna is behind the other. The phase-reference center is used for antenna-pointing direction.

Antennas that are not part of the observation, but still physically present and shadowing other antennas that are being used, must be added to the meta-data list in the zarr prior to calling this method.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input dataset.
- **- shadowlimit or tolerance (in m)** (*TBD*) –
- **Returns** –
- **-------** –
- **xds** (*xarray.core.dataset.Dataset*) – Visibility dataset with updated flags

auto_clip	Apply the clip flagging method. Data with values lower than clip_min

continues on next page

Table 8 – continued from previous page

auto_rflag	
auto_tfcrop	
auto_uvbin	
elevation	
extend	
manager_add	Add a new flag variable to the dataset. All flags in the new variable take
manager_list	Add a new flag variable to the dataset. All flags in the new variable are
manager_remove	Remove flag variable from the dataset.
manual_flag	Implements the 'manual' flagging method (equivalent to CASA6's flagdata manual
manual_unflag	Unflags the selected data. Flags corresponding to the selections are unset.
quack	
shadow	

2.2.4 imaging

Imaging subpackage modules

ngcasa.imaging.calc_image_cell_size

calc_image_cell_size(*vis_dataset*, *global_dataset*, *pixels_per_beam=7*)

Calculates the image and and cell size needed for imaging a vis_dataset. It uses the perfectly-illuminated circular aperture approximation to determine the field of view and pixels_per_beam for the cell size.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input visibility dataset.
- **global_dataset** (*xarray.core.dataset.Dataset*) – Input global dataset (needed for antenna diameter).

Returns

- **imsize** (*list of ints*) – Number of pixels for each spatial dimension.
- **cell** (*list of ints, units = arcseconds*) – Cell size.

ngcasa.imaging.make_grid

make_grid(*vis_mxds*, *img_xds*, *grid_parms*, *vis_sel_parms*, *img_sel_parms*)

Parameters

- **vis_mxds** (*xarray.core.dataset.Dataset*) – Input multi-xarray Dataset with global data.
- **img_xds** (*xarray.core.dataset.Dataset*) – Input image dataset.
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.
- **vis_sel_parms** (*dictionary*) –
- **vis_sel_parms['xds']** (*str*) – The xds within the mxds to use to calculate the imaging weights for.
- **vis_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the xds to use.
- **img_sel_parms** (*dictionary*) –
- **img_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the image xds to use.
- **img_sel_parms['image']** (*str, default ='IMAGE'*) – The created image name.
- **img_sel_parms['sum_weight']** (*str, default ='SUM_WEIGHT'*) – The created sum of weights name.

Returns **img_xds** – The image_dataset will contain the image created and the sum of weights.

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_gridding_convolution_function

make_gridding_convolution_function(*mxds*, *gcf_parms*, *grid_parms*, *sel_parms*)

Currently creates a gcf to correct for the primary beams of antennas and supports heterogenous arrays (antennas with different dish sizes). Only the airy disk and ALMA airy disk model is implemented. In the future support will be added for beam squint, pointing corrections, w projection, and including a prolate spheroidal term.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input visibility dataset.
- **gcf_parms** (*dictionary*) –

- **gcf_parms['function']** (*{'casa_airy'/'airy'}, default = 'casa_airy'*) – The primary beam model used (a function of the dish diameter and blockage diameter).
- **gcf_parms['list_dish_diameters']** (*list of number, units = meter*) – A list of unique antenna dish diameters.
- **gcf_parms['list_blockage_diameters']** (*list of number, units = meter*) – A list of unique feed blockage diameters (must be the same length as gcf_parms['list_dish_diameters']).
- **gcf_parms['unique_ant_indx']** (*list of int*) – A list that has indeces for the gcf_parms['list_dish_diameters'] and gcf_parms['list_blockage_diameters'] lists, for each antenna.
- **gcf_parms['image_phase_center']** (*list of number, length = 2, units = radians*) – The mosaic image phase center.
- **gcf_parms['a_chan_num_chunk']** (*int, default = 3*) – The number of chunks in the channel dimension of the gridding convolution function data variable.
- **gcf_parms['oversampling']** (*list of int, length = 2, default = [10,10]*) – The oversampling of the gridding convolution function.
- **gcf_parms['max_support']** (*list of int, length = 2, default = [15,15]*) – The maximum allowable support of the gridding convolution function.
- **gcf_parms['support_cut_level']** (*number, default = 0.025*) – The antennuation at which to truncate the gridding convolution function.
- **gcf_parms['chan_tolerance_factor']** (*number, default = 0.005*) – It is the fractional bandwidth at which the frequency dependence of the primary beam can be ignored and determines the number of frequencies for which to calculate a gridding convolution function. Number of channels equals the fractional bandwidth devided by gcf_parms['chan_tolerance_factor'].
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.

Returns gcf_dataset

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_image

make_image(*vis_mxds*, *img_xds*, *grid_parms*, *vis_sel_parms*, *img_sel_parms*)

Creates a cube or continuum dirty image from the user specified visibility, uvw and imaging weight data. Only the prolate spheroidal convolutional gridding function is supported. See make_image_with_gcf function for creating an image with A-projection.

Parameters

- **vis_mxds** (*xarray.core.dataset.Dataset*) – Input multi-xarray Dataset with global data.
- **img_xds** (*xarray.core.dataset.Dataset*) – Input image dataset.

- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.
- **vis_sel_parms** (*dictionary*) –
- **vis_sel_parms['xds']** (*str*) – The xds within the mxds to use to calculate the imaging weights for.
- **vis_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the xds to use.
- **img_sel_parms** (*dictionary*) –
- **img_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the image xds to use.
- **img_sel_parms['image']** (*str, default ='IMAGE'*) – The created image name.
- **img_sel_parms['sum_weight']** (*str, default ='SUM_WEIGHT'*) – The created sum of weights name.

Returns **img_xds** – The image_dataset will contain the image created and the sum of weights.

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_image_with_gcf

make_image_with_gcf(*mxds*, *gcf_dataset*, *img_dataset*, *grid_parms*, *norm_parms*, *vis_sel_parms*, *img_sel_parms*)

Creates a cube or continuum dirty image from the user specified visibility, uvw and imaging weight data. A gridding convolution function (gcf_dataset), primary beam image (img_dataset) and a primary beam weight image (img_dataset) must be supplied.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input visibility dataset.
- **gcf_dataset** (*xarray.core.dataset.Dataset*) – Input gridding convolution dataset.
- **img_dataset** (*xarray.core.dataset.Dataset*) – Input image dataset.
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.

- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.
- **norm_parms** (*dictionary*) –
- **norm_parms['norm_type']** (*{'none'/'flat_noise'/'flat_sky'}, default = 'flat_sky'*) –

  Gridded (and FT'd) images represent the PB-weighted sky image. Qualitatively it can be approximated as two instances of the PB applied to the sky image (one naturally present in the data and one introduced during gridding via the convolution functions). normtype='flat_noise' : Divide the raw image by sqrt(sel_parms['weight_pb']) so that

  the input to the minor cycle represents the product of the sky and PB. The noise is 'flat' across the region covered by each PB.

  normtype='flat_sky' [Divide the raw image by sel_parms['weight_pb'] so that the input] to the minor cycle represents only the sky. The noise is higher in the outer regions of the primary beam where the sensitivity is low.

  normtype='none' : No normalization after gridding and FFT.
- **sel_parms** (*dictionary*) –
- **sel_parms['uvw']** (*str, default ='UVW'*) – The name of uvw data variable that will be used to grid the visibilities.
- **sel_parms['data']** (*str, default = 'DATA'*) – The name of the visibility data to be gridded.
- **sel_parms['imaging_weight']** (*str, default ='IMAGING_WEIGHT'*) – The name of the imaging weights to be used.
- **sel_parms['image']** (*str, default ='IMAGE'*) – The created image name.
- **sel_parms['sum_weight']** (*str, default ='SUM_WEIGHT'*) – The created sum of weights name.
- **sel_parms['pb']** (*str, default ='PB'*) – The primary beam image to use for normalization.
- **sel_parms['weight_pb']** (*str, default ='WEIGHT_PB'*) – The primary beam weight image to use for normalization.

Returns **image_dataset** – The image_dataset will contain the image created and the sum of weights.

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_imaging_weight

make_imaging_weight (*vis_mxds*, *imaging_weights_parms*, *grid_parms*, *sel_parms*)

Creates the imaging weight data variable that has dimensions time x baseline x chan x pol (matches the visibility data variable). The weight density can be averaged over channels or calculated independently for each channel using imaging_weights_parms['chan_mode']. The following imaging weighting schemes are supported 'natural', 'uniform', 'briggs', 'briggs_abs'. The grid_parms['image_size'] and grid_parms['cell_size'] should usually be the same values that will be used for subsequent synthesis blocks (for example making the psf). To achieve something similar to 'superuniform' weighting in CASA tclean grid_parms['image_size'] and imaging_weights_parms['cell_size'] can be varied relative to the values used in subsequent synthesis blocks.

Parameters

- **vis_mxds** (*xarray.core.dataset.Dataset*) – Input multi-xarray Dataset with global data.
- **imaging_weights_parms** (*dictionary*) –
- **imaging_weights_parms['weighting']** (*{'natural', 'uniform', 'briggs', 'briggs_abs'}, default = natural*) – Weighting scheme used for creating the imaging weights.
- **imaging_weights_parms['robust']** (*number, acceptable range [-2,2], default = 0.5*) – Robustness parameter for Briggs weighting. robust = -2.0 maps to uniform weighting. robust = +2.0 maps to natural weighting.
- **imaging_weights_parms['briggs_abs_noise']** (*number, default=1.0*) – Noise parameter for imaging_weights_parms['weighting']='briggs_abs' mode weighting.
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.
- **sel_parms** (*dictionary*) –
- **sel_parms['xds']** (*str*) – The xds within the mxds to use to calculate the imaging weights for.
- **sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the xds to use.
- **sel_parms['data_group_out_id']** (*int, default = sel_parms['data_group_id']*) – The output data group. The default will append the imaging weight to the input data group.
- **sel_parms['imaging_weight']** (*str, default ='IMAGING_WEIGHT'*) – The name of that will be used for the imaging weight data variable.

Returns **vis_xds** – The vis_xds will contain a new data variable for the imaging weights the name is defined by the input parameter sel_parms['imaging_weight'].

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_mosaic_pb

make_mosaic_pb(*mxds*, *gcf_dataset*, *img_dataset*, *vis_sel_parms*, *img_sel_parms*, *grid_parms*)

The make_pb function currently supports rotationally symmetric airy disk primary beams. Primary beams can be generated for any number of dishes. The make_pb_parms['list_dish_diameters'] and make_pb_parms['list_blockage_diameters'] must be specified for each dish.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input visibility dataset.
- **gcf_dataset** (*xarray.core.dataset.Dataset*) – Input gridding convolution function dataset.
- **img_dataset** (*xarray.core.dataset.Dataset*) – Input image dataset. ()
- **make_pb_parms** (*dictionary*) –
- **make_pb_parms['function']** (*{'airy'}, default='airy'*) – Only the airy disk function is currently supported.
- **grid_parms['imsize']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **make_pb_parms['list_dish_diameters']** (*list of number*) – The list of dish diameters.
- **= list of number** (*make_pb_parms['list_blockage_diameters']*) – The list of blockage diameters for each dish.
- **vis_sel_parms** (*dictionary*) –
- **vis_sel_parms['xds']** (*str*) – The xds within the mxds to use to calculate the imaging weights for.
- **vis_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the xds to use.
- **img_sel_parms** (*dictionary*) –
- **img_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the image xds to use.
- **img_sel_parms['pb']** (*str, default ='PB'*) – The mosaic primary beam.
- **img_sel_parms['weight_pb']** (*str, default ='WEIGHT_PB'*) – The weight image.
- **img_sel_parms['weight_pb_sum_weight']** (*str, default ='WEIGHT_PB_SUM_WEIGHT'*) – The sum of weight calculated when gridding the gcfs to create the weight image.

Returns img_xds

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_pb

make_pb (*img_xds*, *pb_parms*, *grid_parms*, *sel_parms*)

The make_pb function currently supports rotationally symmetric airy disk primary beams. Primary beams can be generated for any number of dishes. The make_pb_parms['list_dish_diameters'] and make_pb_parms['list_blockage_diameters'] must be specified for each dish.

Parameters

- **img_xds** (*xarray.core.dataset.Dataset*) – Input image dataset.
- **pb_parms** (*dictionary*) –
- **pb_parms['list_dish_diameters']** (*list of number*) – The list of dish diameters.
- **= list of number** (*pb_parms['list_blockage_diameters']*) – The list of blockage diameters for each dish.
- **pb_parms['function']** (*{'casa_airy','airy'}, default='casa_airy'*) – Only the airy disk function is currently supported.
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.
- **sel_parms** (*dictionary*) –
- **= 'PB'** (*sel_parms['pb']*) – The created PB name.

Returns img_xds

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_psf

make_psf (*vis_mxds*, *img_xds*, *grid_parms*, *vis_sel_parms*, *img_sel_parms*)

Creates a cube or continuum point spread function (psf) image from the user specified uvw and imaging weight data. Only the prolate spheroidal convolutional gridding function is supported (this will change in a future releases.)

Parameters

- **vis_mxds** (*xarray.core.dataset.Dataset*) – Input multi-xarray Dataset with global data.
- **img_xds** (*xarray.core.dataset.Dataset*) – Input image dataset.
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).

- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.
- **vis_sel_parms** (*dictionary*) –
- **vis_sel_parms['xds']** (*str*) – The xds within the mxds to use to calculate the imaging weights for.
- **vis_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the xds to use.
- **img_sel_parms** (*dictionary*) –
- **img_sel_parms['data_group_in_id']** (*int, default = first id in xds.data_groups*) – The data group in the image xds to use.
- **img_sel_parms['psf']** (*str, default ='PSF'*) – The created image name.
- **img_sel_parms['psf_sum_weight']** (*str, default ='PSF_SUM_WEIGHT'*) – The created sum of weights name.

Returns **img_xds** – The image_dataset will contain the image created and the sum of weights.

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_psf_with_gcf

make_psf_with_gcf(*mxds*, *gcf_dataset*, *img_dataset*, *grid_parms*, *norm_parms*, *vis_sel_parms*, *img_sel_parms*)

Creates a cube or continuum dirty image from the user specified visibility, uvw and imaging weight data. A gridding convolution function (gcf_dataset), primary beam image (img_dataset) and a primary beam weight image (img_dataset) must be supplied.

Parameters

- **vis_dataset** (*xarray.core.dataset.Dataset*) – Input visibility dataset.
- **gcf_dataset** (*xarray.core.dataset.Dataset*) – Input gridding convolution dataset.
- **img_dataset** (*xarray.core.dataset.Dataset*) – Input image dataset.
- **grid_parms** (*dictionary*) –
- **grid_parms['image_size']** (*list of int, length = 2*) – The image size (no padding).
- **grid_parms['cell_size']** (*list of number, length = 2, units = arcseconds*) – The image cell size.
- **grid_parms['chan_mode']** (*{'continuum'/'cube'}, default = 'continuum'*) – Create a continuum or cube image.
- **grid_parms['fft_padding']** (*number, acceptable range [1,100], default = 1.2*) – The factor that determines how much the gridded visibilities are padded before the fft is done.

- **norm_parms** (*dictionary*) –
- **norm_parms['norm_type']** ({'none'/'flat_noise'/'flat_sky'}, *default = 'flat_sky'*) –

  Gridded (and FT'd) images represent the PB-weighted sky image. Qualitatively it can be approximated as two instances of the PB applied to the sky image (one naturally present in the data and one introduced during gridding via the convolution functions). normtype='flat_noise' : Divide the raw image by sqrt(sel_parms['weight_pb']) so that

  the input to the minor cycle represents the product of the sky and PB. The noise is 'flat' across the region covered by each PB.

  normtype='flat_sky' [Divide the raw image by sel_parms['weight_pb'] so that the input] to the minor cycle represents only the sky. The noise is higher in the outer regions of the primary beam where the sensitivity is low.

  normtype='none' : No normalization after gridding and FFT.

- **sel_parms** (*dictionary*) –
- **sel_parms['uvw']** (*str, default ='UVW'*) – The name of uvw data variable that will be used to grid the visibilities.
- **sel_parms['data']** (*str, default = 'DATA'*) – The name of the visibility data to be gridded.
- **sel_parms['imaging_weight']** (*str, default ='IMAGING_WEIGHT'*) – The name of the imaging weights to be used.
- **sel_parms['image']** (*str, default ='IMAGE'*) – The created image name.
- **sel_parms['sum_weight']** (*str, default ='SUM_WEIGHT'*) – The created sum of weights name.
- **sel_parms['pb']** (*str, default ='PB'*) – The primary beam image to use for normalization.
- **sel_parms['weight_pb']** (*str, default ='WEIGHT_PB'*) – The primary beam weight image to use for normalization.

Returns **image_dataset** – The image_dataset will contain the image created and the sum of weights.

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_sd_image

make_sd_image (*vis_dataset*, *sd_parms*, *storage_parms*)

Todo: This function is not yet implemented

Construct an observed single dish image cube from single-dish data

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_sd_psf

make_sd_psf(*vis_dataset*, *sd_parms*, *storage_parms*)

Todo: This function is not yet implemented

Construct a single dish PSF image cube, containing the effective SD beam per frequency.

Returns **img_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.imaging.make_sd_weight_image

make_sd_weight_image(*vis_dataset*, *sd_parms*, *storage_parms*)

Todo: This function is not yet implemented

Construct a single dish weight map that illustrates the observing pattern of the mosaic.

Returns **vis_dataset**

Return type xarray.core.dataset.Dataset

ngcasa.imaging.predict_modelvis_component

predict_modelvis_component(*img_dataset*, *vis_dataset*, *component_parms*, *storage_parms*)

Todo: This function is not yet implemented

Predict model visibilities from a component list by analytical evaluation

Apply PB models to the components prior to evaluation.

Save the model visibilities in arr_name (default = 'MODEL')

Optionally overwrite the model or add to existing model (incremental=T)

ngcasa.imaging.predict_modelvis_image

predict_modelvis_image(*img_dataset*, *vis_dataset*, *grid_parms*, *storage_parms*)

Todo: This function is not yet implemented

Predict model visibilities from an input model image cube (units Jy/pixel) using a pre-specified gridding convolution function cache.

Save the model visibilities in arr_name (default = 'MODEL')

Optionally overwrite the model or add to existing model (incremental=T)

(A input cube with 1 channel is a continuum image (nterms=1))

Returns **vis_dataset**

Return type xarray.core.dataset.Dataset

calc_image_cell_size	Calculates the image and and cell size needed for imaging a vis_dataset.
make_grid	**param vis_mxds** Input multi-xarray Dataset with global data.
make_gridding_convolution_function	Currently creates a gcf to correct for the primary beams of antennas and supports heterogenous arrays (antennas with different dish sizes).
make_image	Creates a cube or continuum dirty image from the user specified visibility, uvw and imaging weight data. Only the prolate spheroidal convolutional gridding function is supported. See make_image_with_gcf function for creating an image with A-projection.
make_image_with_gcf	Creates a cube or continuum dirty image from the user specified visibility, uvw and imaging weight data. A gridding convolution function (gcf_dataset), primary beam image (img_dataset) and a primary beam weight image (img_dataset) must be supplied.
make_imaging_weight	Creates the imaging weight data variable that has dimensions time x baseline x chan x pol (matches the visibility data variable).
make_mosaic_pb	The make_pb function currently supports rotationally symmetric airy disk primary beams. Primary beams can be generated for any number of dishes.
make_pb	The make_pb function currently supports rotationally symmetric airy disk primary beams. Primary beams can be generated for any number of dishes.
make_psf	Creates a cube or continuum point spread function (psf) image from the user specified uvw and imaging weight data. Only the prolate spheroidal convolutional gridding function is supported (this will change in a future releases.)
make_psf_with_gcf	Creates a cube or continuum dirty image from the user specified visibility, uvw and imaging weight data. A gridding convolution function (gcf_dataset), primary beam image (img_dataset) and a primary beam weight image (img_dataset) must be supplied.
make_sd_image	
make_sd_psf	
make_sd_weight_image	
predict_modelvis_component	

continues on next page

Table 9 – continued from previous page

predict_modelvis_image

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/data_structures.ipynb

CHAPTER

THREE

DATA STRUCTURES

An overview of the new image and visibility data structures and how to convert from the casacore based file formats.

[1]:
```
# Installation
import os
print("installing casa6 + cngi (takes a minute or two)...")
os.system("apt-get install libgfortran3")
os.system("pip install casatasks==6.3.0.48")
os.system("pip install casadata")
os.system("pip install cngi-prototype==0.0.91")

# Retrieve and extract demonstration datasets
print('retrieving MS tarfiles...')
!gdown -q --id 15HfB4rJKqEH7df088Ge5YLrCTXBIax6R
!gdown -q --id 1N9QSs2Hbhi-BrEHx5PA54WigXt8GGgx1
print('extracting MS tarfiles...')
!tar -xf M100.ms.tar
!tar -xzf sis14_twhya_calibrated_flagged.ms.tar.gz

print('complete')
```

```
installing casa6 + cngi (takes a minute or two)...
retrieving MS tarfiles...
extracting MS tarfiles...
complete
```

3.1 MeasurementSet Conversion

CNGI uses an xarray dataset and the zarr storage format to hold the contents of the MS. This provides several advantages to the old MS structure including: 1. Easier access and manipulation of data with numpy-like mathematics 2. N-dim visibility cubes (time, baseline, chan, pol) instead of interlaced rows of variable shape 3. Natively parallel and scalable operations

3.1.1 Data Description IDs

The conversion process will translate an MS directory on disk into one or more Xarray Datasets (xds) and (optionally) store them to a Zarr directory on disk. To properly support dimension expansion from interlaced rows, each MS SPW/Pol combination, denoted by Data Description ID (**DDI**), must be processed and partitioned individually, along with the various subtables.

We begin by inspecting the structure of the M100 MeasurementSet to determine how best to convert it

[2]:
```
from cngi.conversion import describe_ms

describe_ms('M100.ms')
```

[2]:
```
     spw_id  pol_id   rows  times  baselines  chans  pols  size_MB
ddi
0         0       0  32464    568        154    240     2      361
1         1       0  32464    568        154    240     2      361
2         2       0  32464    568        154    240     2      361
3         3       0  31408    568        142    240     2      333
```

This demonstration MS has four DDI's, corresponding to four different SPW's. In this case they are of similar modest size.

Let's convert two of them now using the default settings

[3]:
```
from cngi.conversion import convert_ms

mxds = convert_ms('M100.ms', ddis=[0,2])
```

```
Completed ddi 0  process time 10.62 s
Completed ddi 2  process time 10.09 s
```

Larger numbers of baselines and channels will consume more memory during conversion. If a particular DDI shape is too large for the host machine memory, a smaller chunk size along the time axis may be needed.

Let's pretend this is the case for the other two DDI's and use a smaller chunk size for them, making sure to append and not overwrite our first two

[4]:
```
mxds = convert_ms('M100.ms', ddis=[1,3], chunks=(50,400,32,1), append=True)
```

```
Completed ddi 1  process time 12.99 s
Completed ddi 3  process time 12.24 s
```

Finally, lets get the subtables within the original MS. They are now referred to as the "global" data in Zarr directory as their contents applies to all of the DDI partitions.

Some subtable columns may give errors during conversion if the casacore table system cannot read them as numpy arrays. The resulting Xarray dataset will omit these columns. *This is a known issue*

[5]:
```
mxds = convert_ms('M100.ms', ddis=['global'], append=True)
```

```
Completed subtables  process time 0.87 s
```

If we are comfortable with leaving things at default, we can convert the entire contents of an MS at once. The TWHya MS is small and safe to convert without much worry, but we will give it a shorter output filename.

[6]:
```
from cngi.conversion import convert_ms

mxds = convert_ms('sis14_twhya_calibrated_flagged.ms', outfile='twhya.vis.zarr')
```

```
Completed ddi 0  process time 23.69 s
Completed subtables  process time 1.21 s
```

3.1.2 Xarray/Zarr Partitions

After converting an MS to the new Xarray / Zarr based format, we have a `<filename>.vis.zarr` directory on disk. We can inspect its contents to see if the conversion was successful.

[7]:
```
from cngi.dio import describe_vis

describe_vis('M100.vis.zarr')
```

[7]:

	spw_id	pol_id	times	baselines	chans	pols	size_MB
xds							
xds0	0	0	568	154	240	2	1011
xds1	1	0	568	154	240	2	1011
xds2	2	0	568	154	240	2	1011
xds3	3	0	568	142	240	2	932

The four SPW's are now contained in four separate partitions. There are no more rows, only the four dimensions of (time, baseline, channel, polarization) to describe each field.

When we go to open and use the new format for subsequent CNGI operations, we will refer to the specific visibility xarray dataset (**xds**) that we want to use.

3.1.3 MeasurementSet v3 Schema

The conversion process attempts to keep the same column names, definitions, and relationships from the original MS structure whenever possible. This means that, for example, the DATA column of the main table in the MS is still called DATA in the Xarray Dataset, but it is now a data variable within the dataset. Similarly the column names of the various subtables are reflected as data variable names in the new xarray datasets.

As part of the evolution to new datastructures, the opportunity was taken to also update to the new MSv3 schema. This means that when converting the current CASA MSv2 datasets, certain columns are translated or dropped per the MSv3 definition located here.

While CNGI typically only operates on a single visibility partition, we can open and inspect the entire zarr directory contents in a manner similar to how the entire MS could be opened and inspected previously. This is done by constructing an xarray *dataset of datasets*, referred to as the master xarray dataset (**mxds**).

[8]:
```
from cngi.dio import read_vis

mxds = read_vis('M100.vis.zarr')

print(mxds)
```

```
overwrite_encoded_chunks True
<xarray.Dataset>
Dimensions:           (antenna_ids: 27, feed_ids: 108, field_ids: 48, observation_ids:
↪ 4, polarization_ids: 1, source_ids: 4, spw_ids: 4, state_ids: 24)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * antenna_ids       (antenna_ids) int64 0 1 2 3 4 5 6 ... 20 21 22 23 24 25 26
    antennas          (antenna_ids) <U16 'CM01' 'DV01' 'DV03' ... 'PM01' 'PM03'
  * field_ids         (field_ids) int64 0 1 2 3 4 5 6 7 ... 41 42 43 44 45 46 47
    fields            (field_ids) <U16 'M100' 'M100' 'M100' ... 'M100' 'M100'
  * feed_ids          (feed_ids) int64 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  * observation_ids   (observation_ids) int64 0 1 2 3
    observations      (observation_ids) <U16 'T.B.D.' 'T.B.D.' 'T.B.D.' 'T.B.D.'
  * polarization_ids  (polarization_ids) int64 0
  * source_ids        (source_ids) int64 0 0 0 0
    sources           (source_ids) <U16 'M100' 'M100' 'M100' 'M100'
  * spw_ids           (spw_ids) int64 0 1 2 3
  * state_ids         (state_ids) int64 0 1 2 3 4 5 6 7 ... 17 18 19 20 21 22 23
Data variables:
    *empty*
Attributes: (12/17)
    xds2:             <xarray.Dataset>\nDimensions:         (baseline: 154, c...
    xds3:             <xarray.Dataset>\nDimensions:         (baseline: 142, c...
    xds0:             <xarray.Dataset>\nDimensions:         (baseline: 154, c...
    xds1:             <xarray.Dataset>\nDimensions:         (baseline: 154, c...
    ANTENNA:          <xarray.Dataset>\nDimensions:        (antenna_id: 27, d...
    ASDM_ANTENNA:     <xarray.Dataset>\nDimensions:         (d0: 13, d1: 3)\n...
    ...               ...
    POLARIZATION:     <xarray.Dataset>\nDimensions:       (d0: 1, d1: 2, d2: ...
    PROCESSOR:        <xarray.Dataset>\nDimensions:   (d0: 3)\nCoordinates:\n...
    SOURCE:           <xarray.Dataset>\nDimensions:             (d0: 4, d1: 2...
    SPECTRAL_WINDOW:  <xarray.Dataset>\nDimensions:             (d1: 240, d2:...
    STATE:            <xarray.Dataset>\nDimensions:   (state_id: 24)\nCoordin...
    WEATHER:          <xarray.Dataset>\nDimensions:                (d0: 409,...
```

The mxds **coordinates** describe the principal keys to different tables in the MSv3 schema. The **attributes** section holds references to each individual xarray dataset visibility partition and subtable of global data.

Inspecting the FIELD subtable shows fields matching the same columns as the MSv3 schema.

[9]:
```
print(mxds.FIELD)
```

```
<xarray.Dataset>
Dimensions:         (d1: 1, d2: 2, field_id: 48)
Coordinates:
  * field_id        (field_id) int64 0 1 2 3 4 5 6 7 ... 40 41 42 43 44 45 46 47
    source_id       (field_id) int64 dask.array<chunksize=(48,), meta=np.ndarray>
Dimensions without coordinates: d1, d2
Data variables:
    CODE            (field_id) <U16 dask.array<chunksize=(48,), meta=np.ndarray>
    DELAYDIR_REF    (field_id) int64 dask.array<chunksize=(48,), meta=np.ndarray>
    DELAY_DIR       (field_id, d1, d2) float64 dask.array<chunksize=(48, 1, 2),␣
↪meta=np.ndarray>
    NAME            (field_id) <U16 dask.array<chunksize=(48,), meta=np.ndarray>
    NUM_POLY        (field_id) int64 dask.array<chunksize=(48,), meta=np.ndarray>
    PHASEDIR_REF    (field_id) int64 dask.array<chunksize=(48,), meta=np.ndarray>
    PHASE_DIR       (field_id, d1, d2) float64 dask.array<chunksize=(48, 1, 2),␣
↪meta=np.ndarray>
    REFDIR_REF      (field_id) int64 dask.array<chunksize=(48,), meta=np.ndarray>
    REFERENCE_DIR   (field_id, d1, d2) float64 dask.array<chunksize=(48, 1, 2),␣
↪meta=np.ndarray>
    TIME            (field_id) datetime64[ns] dask.array<chunksize=(48,), meta=np.
↪ndarray>
```

The main table of the MSv3 schema has been divided in to the four visibility xarray dataset (xds) partitions that CNGI functions operate on. Inspecting an xds partition shows fields that correspond to the columns of the main table. A difference arises from the expansion of **time** and **baseline** dimensions from what used to be **rows**.

Note that in cases where a different number of baselines exist at each time step within a single SPW of the originating MS, the resulting XDS will have the maximum number of baselines set in that dimension with NaN padding added as necessary.

3.2 Visibility Dataset Structure

The visibility xarray dataset (xds) structure has four main components: 1. dimensions 2. coordinates 3. data variables 4. attributes

Dimensions define the shape of the other components, and allow indexing into other components by integer location within each dimension (ie channel 5). Note that dimensions may be printed alphabetically by Jupyter, with the actual order being different in the data itself. Referring to a dimension by its name eliminates the need to remember what order things are in.

Coordinates define the world values of dimensions and other indices within the dataset. This allows indexing into other components by actual value (ie channel 100 GHz). Note that in many cases the real world value is itself just an integer index (ie the baseline), but time and channel frequency are particularly useful.

Data variables are the columns of the main table. They typically have the same data type and meaning as defined in the MSv3 schema. They are stored as Dask arrays which allow numpy-like operations that are parallel, scalable, and support larger than memory data sizes. Nan values are used to pad and flag areas with no valid data, and consequently all mathematics must be smart enough to properly ignore Nans in computations.

Attributes are used to hold units, reference frames, and any other metadata associated with the dataset. They can be any python type or object when in memory, but only serializable types may be written to disk.

Each xds comes from a separate zarr partition, and corresponds to a particular spw and polarization combination (as denoted by coordinate values). Here we can see another of the four xds paritions from the previous conversion of the demonstration MS

[10]:
```
print(mxds.xds0)
```

```
<xarray.Dataset>
Dimensions:          (baseline: 154, chan: 240, pol: 2, pol_id: 1, spw_id: 1, time:␣
↪568, uvw_index: 3)
Coordinates:
  * baseline         (baseline) int64 0 1 2 3 4 5 6 ... 148 149 150 151 152 153
  * chan             (chan) float64 1.137e+11 1.137e+11 ... 1.156e+11 1.156e+11
    chan_width       (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
    effective_bw     (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
  * pol              (pol) int64 9 12
  * pol_id           (pol_id) int64 0
    resolution       (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
  * spw_id           (spw_id) int64 0
  * time             (time) datetime64[ns] 2011-08-10T19:38:17.856000900 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1         (baseline) int64 dask.array<chunksize=(154,), meta=np.ndarray>
    ANTENNA2         (baseline) int64 dask.array<chunksize=(154,), meta=np.ndarray>
    ARRAY_ID         (time, baseline) int64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    DATA             (time, baseline, chan, pol) complex128 dask.array<chunksize=(100,␣
↪154, 32, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    DATA_WEIGHT     (time, baseline, chan, pol) float64 dask.array<chunksize=(100,␣
↪154, 32, 1), meta=np.ndarray>
    EXPOSURE        (time, baseline) float64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    ...              ...
    OBSERVATION_ID  (time, baseline) int64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    PROCESSOR_ID    (time, baseline) int64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    SCAN_NUMBER     (time, baseline) int64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    STATE_ID        (time, baseline) int64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    TIME_CENTROID   (time, baseline) float64 dask.array<chunksize=(100, 154), meta=np.
↪ndarray>
    UVW             (time, baseline, uvw_index) float64 dask.array<chunksize=(100,␣
↪154, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           1
    corr_product:     [[0, 0], [1, 1]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:
    net_sideband:     2
    num_chan:         240
    num_corr:         2
    ref_frequency:    113730081250.0
    total_bandwidth:  1875000000.0
```

The chunk size of the Dask array based Data variables affects the smallest unit of data on disk that may be loaded and processed by a worker. A larger number of smaller chunks provides more units of work to go around in a parallel processing environment, with less memory needed for each worker. However, an overhead cost of scheduling and managing each unit of work creates a point of diminishing returns.

We converted partitions 0 and 2 differently than partitions 1 and 3, using a smaller chunksize on the time dimension for the latter two. We can see the effect of this on the dask structure

[11]:
```
mxds.xds0.DATA.chunks
```

[11]:
```
((100, 100, 100, 100, 100, 68),
 (154,),
 (32, 32, 32, 32, 32, 32, 32, 16),
 (1, 1))
```

[12]:
```
mxds.xds1.DATA.chunks
```

[12]:
```
((50, 50, 50, 50, 50, 50, 50, 50, 50, 50, 18),
 (154,),
 (32, 32, 32, 32, 32, 32, 32, 16),
 (1, 1))
```

3.3 Image Conversion

Image data is converted and stored in a manner very similar to MeasurementSet data. The same xarray and zarr frameworks are used, with primary difference being the schema layout of contents within the xarray dataset structure.

Images may have a number of supporting products (residual, pb, psf, taylor terms, masks, etc) in their own separate directories. The xarray dataset and the zarr storage format is capable of storing all image products (of the same shape) together. The resulting single CNGI data structure is more convenient to work with than the original disparate directories.

As a convenience, the conversion routines return an xarray data structure (xds) object attached to the zarr directory holding the converted data. The same xds can be retrieved later using the `read_image()` function, so conversion is only necessary once.

3.3.1 Create images with tclean

First we need to create some images from the downloaded MeasurementSet using the CASA6 tclean task. We will create a cube image, a continuum image, and an MTMFS image with Taylor terms.

[13]:
```
!rm -fr twhya_*
```

[14]:
```
from casatasks import tclean

print('creating cube')
tclean(vis='sis14_twhya_calibrated_flagged.ms', imagename='twhya_cube', field='5',␣
↪spw='',
       specmode='cube', cell=1, pbcor=True, imsize=100, nchan=10, deconvolver='hogbom
↪',
       niter=10, savemodel='modelcolumn', usemask='auto-multithresh')

print('creating continuum')
tclean(vis='sis14_twhya_calibrated_flagged.ms', imagename='twhya_cont', field='5',␣
↪spw='',
       specmode='mfs', cell=1, pbcor=True, imsize=100, deconvolver='hogbom',
       niter=10, savemodel='modelcolumn', usemask='auto-multithresh')

print('creating mtmfs')
tclean(vis='sis14_twhya_calibrated_flagged.ms', imagename='twhya_mtmfs', field='5',␣
↪spw='',
       specmode='mfs', cell=1, pbcor=True, imsize=100, deconvolver='mtmfs',
       nterms=5, niter=10, savemodel='modelcolumn', usemask='auto-multithresh')

print('complete')
```

```
creating cube
creating continuum
creating mtmfs
complete
```

3.3.2 Cube Images

The previous tclean call produced a cube image with 10 channels along with a number of supporting image products (same filename with a different extension). Some of these image products have additional embedded masks.

[15]:
```
!ls -d twhya_cube.*
```

```
twhya_cube.image       twhya_cube.mask   twhya_cube.pb   twhya_cube.residual
twhya_cube.image.pbcor twhya_cube.model  twhya_cube.psf  twhya_cube.sumwt
```

The CNGI conversion function will merge these individual products and any embedded masks into a single xarray dataset stored in a single img.zarr directory.

Note that the .mask image product will be renamed to 'automask' in the xarray dataset

[16]:
```
from cngi.conversion import convert_image

xds = convert_image('twhya_cube.image')

print(xds.dims)
print(xds.data_vars)
```

```
converting Image...
processed image in 1.0922942 seconds
Frozen(SortedKeysDict({'l': 100, 'm': 100, 'time': 1, 'chan': 10, 'pol': 1}))
Data variables:
    AUTOMASK           (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB                 (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0           (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

3.3.3 Continuum Images

As with cube images, the continuum image products are similar and merged in a similar manner. The main difference in the output is of course the number of channels

[17]:
```
xds = convert_image('twhya_cont.image')

print(xds.dims)
print(xds.data_vars)
```

```
converting Image...
processed image in 0.64222145 seconds
Frozen(SortedKeysDict({'l': 100, 'm': 100, 'time': 1, 'chan': 1, 'pol': 1}))
Data variables:
    AUTOMASK           (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB                 (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0           (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

3.3.4 MTMFS Images

Multi-Term Multi-Freqeuency Synthesis images may contain taylor term expansions in place of the channel dimension. In CASA6 these are stored as a number of individual image directories with a .ttN extension.

[18]:
```
!ls -d twhya_mtmfs.*
```

```
twhya_mtmfs.alpha           twhya_mtmfs.model.tt1  twhya_mtmfs.psf.tt8
twhya_mtmfs.alpha.error      twhya_mtmfs.model.tt2  twhya_mtmfs.residual.tt0
twhya_mtmfs.alpha.pbcor      twhya_mtmfs.model.tt3  twhya_mtmfs.residual.tt1
twhya_mtmfs.beta            twhya_mtmfs.model.tt4  twhya_mtmfs.residual.tt2
twhya_mtmfs.beta.pbcor      twhya_mtmfs.pb.tt0     twhya_mtmfs.residual.tt3
twhya_mtmfs.image.tt0       twhya_mtmfs.pb.tt1     twhya_mtmfs.residual.tt4
twhya_mtmfs.image.tt0.pbcor  twhya_mtmfs.pb.tt2     twhya_mtmfs.sumwt.tt0
twhya_mtmfs.image.tt1       twhya_mtmfs.pb.tt3     twhya_mtmfs.sumwt.tt1
twhya_mtmfs.image.tt1.pbcor  twhya_mtmfs.pb.tt4     twhya_mtmfs.sumwt.tt2
twhya_mtmfs.image.tt2       twhya_mtmfs.psf.tt0    twhya_mtmfs.sumwt.tt3
twhya_mtmfs.image.tt2.pbcor  twhya_mtmfs.psf.tt1    twhya_mtmfs.sumwt.tt4
```

(continues on next page)

(continued from previous page)

```
twhya_mtmfs.image.tt3       twhya_mtmfs.psf.tt2    twhya_mtmfs.sumwt.tt5
twhya_mtmfs.image.tt3.pbcor  twhya_mtmfs.psf.tt3    twhya_mtmfs.sumwt.tt6
twhya_mtmfs.image.tt4       twhya_mtmfs.psf.tt4    twhya_mtmfs.sumwt.tt7
twhya_mtmfs.image.tt4.pbcor  twhya_mtmfs.psf.tt5    twhya_mtmfs.sumwt.tt8
twhya_mtmfs.mask            twhya_mtmfs.psf.tt6
twhya_mtmfs.model.tt0       twhya_mtmfs.psf.tt7
```

The CNGI conversion will merged these individual .ttN directories in to the channel dimension of the corresponding image product.

[19]:
```
xds = convert_image('twhya_mtmfs.image')

print(xds.dims)
print(xds.data_vars)
```

```
converting Image...
processed image in 0.97542477 seconds
Frozen(SortedKeysDict({'l': 100, 'm': 100, 'time': 1, 'chan': 5, 'pol': 1}))
Data variables:
    AUTOMASK           (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB                 (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0           (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

3.4 Image Dataset Structure

The image xarray dataset (xds) structure has four main components: 1. dimensions 2. coordinates 3. data variables 4. attributes

Dimensions define the shape of the other components, and allow indexing into other components by integer location within each dimension (ie channel 5). Note that dimensions may be printed alphabetically by Jupyter, with the actual order being different in the data itself. Referring to a dimension by its name eliminates the need to remember what order things are in.

Coordinates define the world values of dimensions and other indices within the dataset. This allows indexing into

other components by actual value (ie channel 100 GHz). Note that coordinates may be multi-dimensional products of underlying dimensions. This allows the storage of spherical right-ascension / declination pairs for each cartesion pixel value in the image (l / m dimensions)

Data variables are the image products and masks. They are stored as Dask arrays which allow numpy-like operations that are parallel, scalable, and support larger than memory data sizes. Nan values are used to mask areas with no valid data, and consequently all mathematics must be smart enough to properly ignore Nans in computations.

Attributes are used to hold units, reference frames, and any other metadata associated with the dataset. They can be any python type or object when in memory, but only serializable types may be written to disk.

A time dimension has been inserted in the converted image data. This is a placeholder for future time-domain image handling in CNGI. CASA6 does not currently produce these types of images.

Inspecting the cube xds shows 10 frequency channels of the original image shared by all the image products

[20]:
```
from cngi.dio import read_image

xds = read_image('twhya_cube.img.zarr')

print(xds)
print('\nChannel Frequencies: ', xds.chan.values)
```

```
<xarray.Dataset>
Dimensions:             (chan: 10, l: 100, m: 100, pol: 1, time: 1)
Coordinates:
  * chan                (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11
    declination         (l, m) float64 dask.array<chunksize=(100, 100), meta=np.
↪ndarray>
  * l                   (l) float64 0.0002424 0.0002376 ... -0.0002327 -0.0002376
  * m                   (m) float64 -0.0002424 -0.0002376 ... 0.0002327 0.0002376
  * pol                 (pol) float64 1.0
    right_ascension     (l, m) float64 dask.array<chunksize=(100, 100), meta=np.
↪ndarray>
  * time                (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    AUTOMASK            (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE               (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0         (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR         (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0   (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL               (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB                  (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0            (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                 (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL            (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0      (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    SUMWT               (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes: (12/19)
    axisnames:             ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:             ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:            [1.2625414355032467, 1.1318049125886214, -84.517746...
    commonbeam_units:      ['arcsec', 'arcsec', 'deg']
    date_observation:      2012/11/19/07
    direction_reference:   j2000
    ...                    ...
    restoringbeam:         [1.2625414355032467, 1.1318049125886214, -84.517746...
    spectral__reference:   lsrk
    telescope:             alma
    telescope_position:    [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                  Jy/beam
    velocity__type:        radio

Channel Frequencies:  [3.72520023e+11 3.72520633e+11 3.72521243e+11 3.72521854e+11
 3.72522464e+11 3.72523074e+11 3.72523685e+11 3.72524295e+11
 3.72524905e+11 3.72525516e+11]
```

The MTMFS image is a continuum with each taylor term stored in the channel dimension. This means that the channel coordinate values will all be the same with the length/index position within the channel dimension corresponding to the taylor polynomial coefficient

[21]:
```
from cngi.dio import read_image

xds = read_image('twhya_mtmfs.img.zarr')

print(xds)
print('\nChannel Frequencies: ', xds.chan.values)
```

```
<xarray.Dataset>
Dimensions:             (chan: 5, l: 100, m: 100, pol: 1, time: 1)
Coordinates:
  * chan                (chan) float64 3.726e+11 3.726e+11 ... 3.726e+11
    declination         (l, m) float64 dask.array<chunksize=(100, 100), meta=np.
↪ndarray>
  * l                   (l) float64 0.0002424 0.0002376 ... -0.0002327 -0.0002376
  * m                   (m) float64 -0.0002424 -0.0002376 ... 0.0002327 0.0002376
  * pol                 (pol) float64 1.0
    right_ascension     (l, m) float64 dask.array<chunksize=(100, 100), meta=np.
↪ndarray>
  * time                (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    AUTOMASK            (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE               (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0         (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR         (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0   (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL               (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    PB                (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0          (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    PSF               (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL          (l, m, time, chan, pol) float64 dask.array<chunksize=(100, 100,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0    (l, m, time, chan, pol) bool dask.array<chunksize=(100, 100, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT             (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes: (12/19)
    axisnames:            ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:            ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:           [1.2610454559326172, 1.131009578704834, -84.4926986...
    commonbeam_units:     ['arcsec', 'arcsec', 'deg']
    date_observation:     2012/11/19/07
    direction_reference:  j2000
    ...                   ...
    restoringbeam:        [1.2610454559326172, 1.131009578704834, -84.4926986...
    spectral__reference:  lsrk
    telescope:            alma
    telescope_position:   [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                 Jy/beam
    velocity__type:       radio

Channel Frequencies:  [3.7263694e+11 3.7263694e+11 3.7263694e+11 3.7263694e+11 3.
↪7263694e+11]
```

The chunk size of the Dask array based Data variables affects the smallest unit of data on disk that may be loaded and processed by a worker. A larger number of smaller chunks provides more units of work to go around in a parallel processing environment, with less memory needed for each worker. However, an overhead cost of scheduling and managing each unit of work creates a point of diminishing returns.

The CNGI conversion function allows for tailoring the chunk size beyond the default values used in this overview.

[22]:
```
xds.IMAGE.chunks
```

[22]:
```
((100,), (100,), (1,), (1, 1, 1, 1, 1), (1,))
```

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/visibilities.ipynb

CHAPTER

FOUR

VISIBILITIES

An overview of the visibility data structure and manipulation.

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here

[1]:
```
# Installation
import os
print("installing casa6 + cngi (takes a minute or two)...")
os.system("apt-get install libgfortran3")
os.system("pip install casatasks==6.3.0.48")
os.system("pip install casadata")
os.system("pip install cngi-prototype==0.0.91")

# Retrieve and extract demonstration dataset
print('retrieving MS tarfile...')
!gdown -q --id 1N9QSs2Hbhi-BrEHx5PA54WigXt8GGgx1
!tar -xzf sis14_twhya_calibrated_flagged.ms.tar.gz
print('complete')
```

```
installing casa6 + cngi (takes a minute or two)...
retrieving MS tarfile...
complete
```

4.1 Initialize the Environment

`InitializeFramework` instantiates a client object (does not need to be returned and saved by caller). Once this object exists, all Dask objects automatically know to use it for parallel execution.

```
>>> from cngi.direct import InitializeFramework
>>> client = InitializeFramework(workers=4, memory='2GB')
>>> print(client)
<Client: 'tcp://127.0.0.1:33227' processes=4 threads=4, memory=8.00 GB>
```

Omitting this step will cause the subsequent Dask dataframe operations to use the default built-in scheduler for parallel execution (which can actually be faster on local machines anyway)

Google Colab doesn't really support the dask.distributed environment particularly well, so we will let Dask use its default scheduler.

4.2 MeasurementSet Conversion

CNGI uses an xarray dataset (xds) and the zarr storage format to hold the contents of the MS. This provides several advantages to the old MS structure including: 1. Easier access and manipulation of data with numpy-like mathematics 2. N-dim visibility cubes (time, baseline, chan, pol) instead of interlaced rows of variable shape 3. Natively parallel and scalable operations

[2]:
```
from cngi.conversion import convert_ms

mxds = convert_ms('sis14_twhya_calibrated_flagged.ms', outfile='twhya.vis.zarr')
```

```
Completed ddi 0  process time 25.09 s
Completed subtables  process time 1.21 s
```

An xarray dataset of datasets (mxds) is used to hold the main table and subtables from the original MS in separate xds structures. The main table is further separated by spw/pol combination in to individual xds structures of fixed shape.

[3]:
```
print(mxds)
```

```
<xarray.Dataset>
Dimensions:            (antenna_ids: 26, feed_ids: 26, field_ids: 7, observation_ids:␣
↪1, polarization_ids: 1, source_ids: 5, spw_ids: 1, state_ids: 20)
Coordinates:
  * antenna_ids        (antenna_ids) int64 0 1 2 3 4 5 6 ... 19 20 21 22 23 24 25
    antennas           (antenna_ids) <U16 'DA41' 'DA42' 'DA44' ... 'DV22' 'DV23'
  * field_ids          (field_ids) int64 0 1 2 3 4 5 6
    fields             (field_ids) <U9 'J0522-364' 'J0539+145' ... '3c279'
  * feed_ids           (feed_ids) int64 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  * observation_ids    (observation_ids) int64 0
    observations       (observation_ids) <U23 'uid://A002/X327408/X6f'
  * polarization_ids   (polarization_ids) int64 0
  * source_ids         (source_ids) int64 0 1 2 3 4
    sources            (source_ids) <U9 'J0522-364' 'Ceres' ... 'TW Hya' '3c279'
  * spw_ids            (spw_ids) int64 0
  * state_ids          (state_ids) int64 0 1 2 3 4 5 6 7 ... 13 14 15 16 17 18 19
Data variables:
    *empty*
Attributes: (12/17)
    xds0:              <xarray.Dataset>\nDimensions:         (baseline: 210, c...
    ANTENNA:           <xarray.Dataset>\nDimensions:        (antenna_id: 26, d...
    ASDM_ANTENNA:      <xarray.Dataset>\nDimensions:         (d0: 26, d1: 3)\n...
    ASDM_CALWVR:       <xarray.Dataset>\nDimensions:            (d0: 702, d1: ...
    ASDM_RECEIVER:     <xarray.Dataset>\nDimensions:           (d0: 26, d1: 2,...
    ASDM_STATION:      <xarray.Dataset>\nDimensions:    (d0: 28, d1: 3)\nDimen...
    ...                ...
    POLARIZATION:      <xarray.Dataset>\nDimensions:       (d0: 1, d1: 2, d2: ...
    PROCESSOR:         <xarray.Dataset>\nDimensions:   (d0: 3)\nCoordinates:\n...
    SOURCE:            <xarray.Dataset>\nDimensions:            (d0: 5, d1: 2...
    SPECTRAL_WINDOW:   <xarray.Dataset>\nDimensions:            (d1: 384, d2:...
    STATE:             <xarray.Dataset>\nDimensions:   (state_id: 20)\nCoordin...
    WEATHER:           <xarray.Dataset>\nDimensions:                (d0: 385,...
```

[4]:
```
print(mxds.xds0)
```

```
<xarray.Dataset>
Dimensions:          (baseline: 210, chan: 384, pol: 2, pol_id: 1, spw_id: 1, time:␣
↪410, uvw_index: 3)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * baseline        (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
  * chan            (chan) float64 3.725e+11 3.725e+11 ... 3.728e+11 3.728e+11
    chan_width      (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
    effective_bw    (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
  * pol             (pol) int64 9 12
  * pol_id          (pol_id) int64 0
    resolution      (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
  * spw_id          (spw_id) int64 0
  * time            (time) datetime64[ns] 2012-11-19T07:37:00 ... 2012-11-19T...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1        (baseline) int64 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2        (baseline) int64 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID        (time, baseline) int64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    DATA            (time, baseline, chan, pol) complex128 dask.array<chunksize=(100,␣
↪210, 32, 1), meta=np.ndarray>
    DATA_WEIGHT     (time, baseline, chan, pol) float64 dask.array<chunksize=(100,␣
↪210, 32, 1), meta=np.ndarray>
    EXPOSURE        (time, baseline) float64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    ...              ...
    OBSERVATION_ID  (time, baseline) int64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    PROCESSOR_ID    (time, baseline) int64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    SCAN_NUMBER     (time, baseline) int64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    STATE_ID        (time, baseline) int64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    TIME_CENTROID   (time, baseline) float64 dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    UVW             (time, baseline, uvw_index) float64 dask.array<chunksize=(100,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/14)
    assoc_nature:     ['', '', '', '', '', '', '', '', '', '', '', '', '', ''...
    bbc_no:           2
    corr_product:     [[0, 0], [1, 1]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         384
    num_corr:         2
    ref_frequency:    372533086425.9812
    total_bandwidth:  234375000.0
```

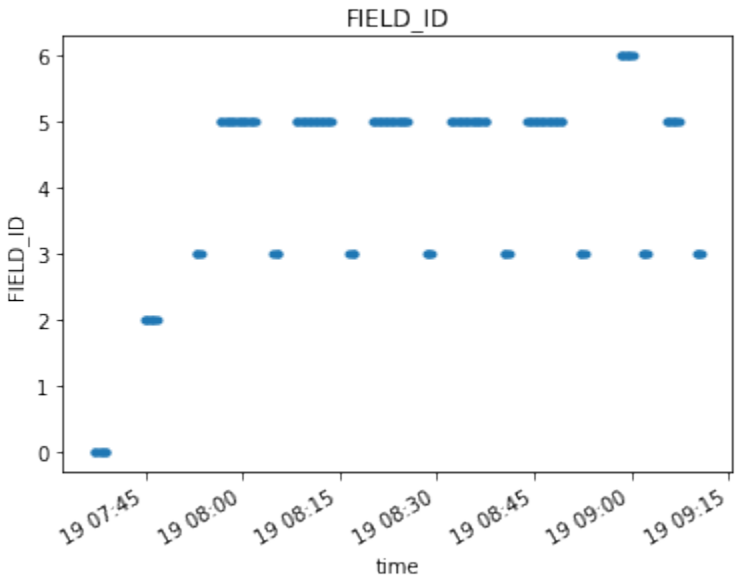
4.3 Simple Plotting

We can quickly spot check data fields using `visplot`. This is handy during subsequent analysis (although not intended for full scientific analysis).

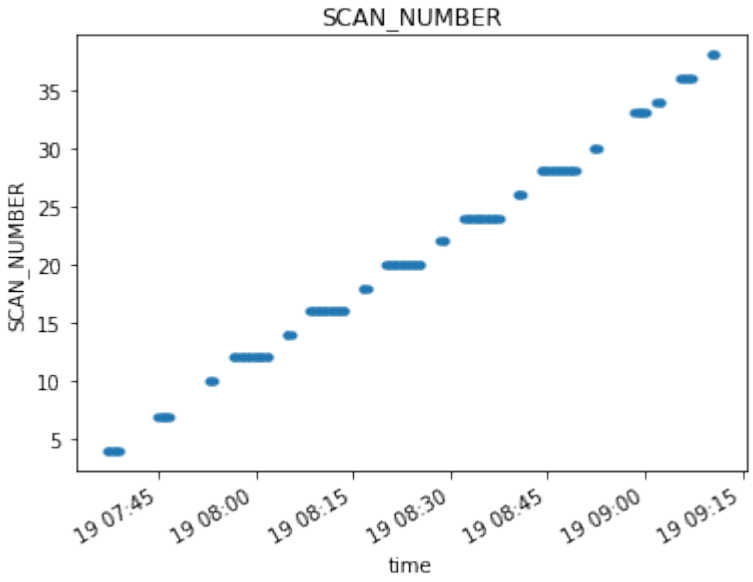
The visplot function supports both standard x-y plots and x-y-c color mesh visualization. The mode is determined by the number of dimensions in the data passed in

[5]:
```
from cngi.dio import read_vis
from cngi.vis import visplot

# open a visibility xds
xds = read_vis('twhya.vis.zarr').xds0

# fields versus time coordinate
visplot(xds.FIELD_ID, axis='time')
```

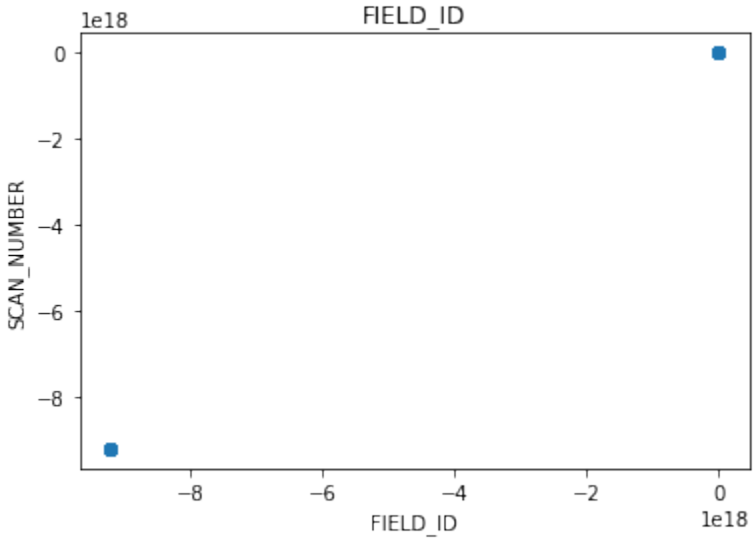
```
overwrite_encoded_chunks True
```



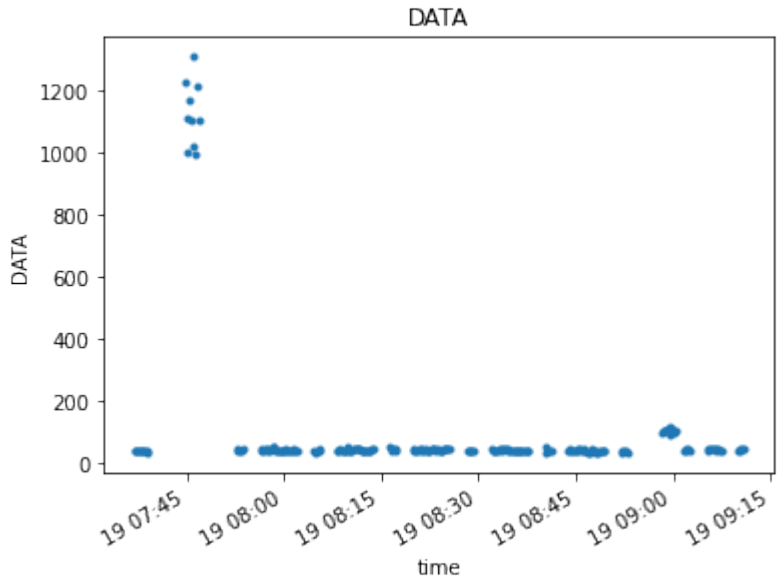
[6]:
```
visplot(xds.SCAN_NUMBER, 'time')
```



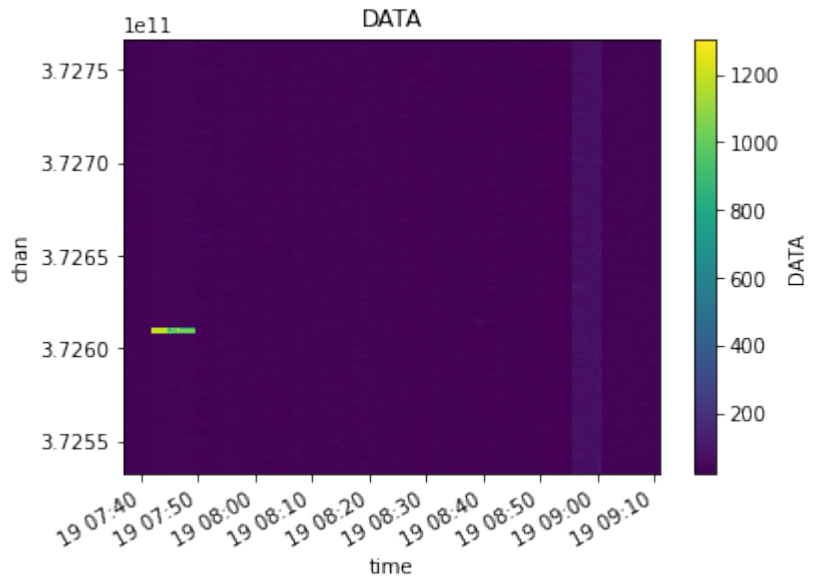
[7]:
```
# 2-D plot - fields versus scans
visplot(xds.FIELD_ID, xds.SCAN_NUMBER)
```



[8]:
```
# 2-D plot of 4-D data - DATA over time
# when we specify an x-axis (time), the other axes (baseline, chan, pol) are averaged␣
↪together
visplot(xds.DATA, axis='time')
```



[9]:
```
# 3-D plot of 4-D data - DATA over time and channel
# we can give two axes to create a color mesh visualization
visplot(xds.DATA, axis=['time','chan'])
```



4.4 Data Selection

The xarray Dataset format has extensive built-in functions for selecting data and splitting by different criteria. Often times the global data is referenced to identify particular values to select by.

Once the selection values are known, `xds.isel(...)` and `xds.sel(...)` can be used to select by dimension index or world value. `xds.where(...)` can be used to select by anything.

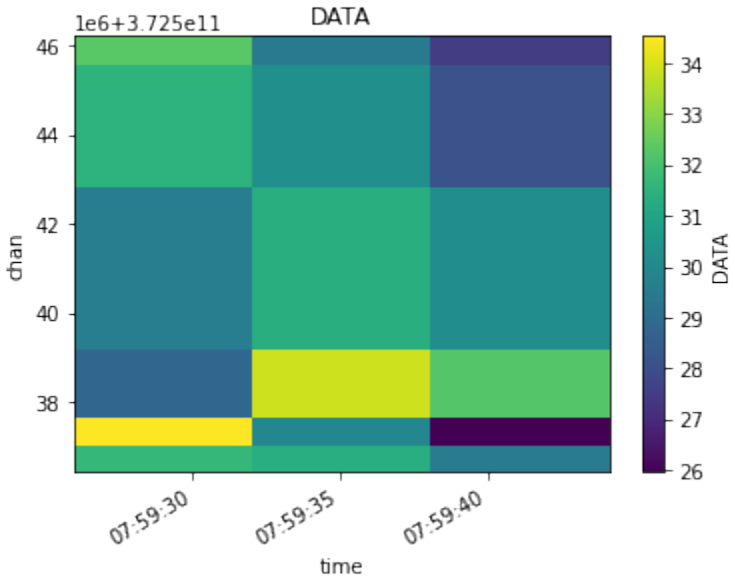
Lets start by examining the range of time and channel frequency values we have available in the visibility xds along with the field names and state ids in the global data.

[10]:
```
from cngi.dio import read_vis
import numpy as np

# open the master xds
mxds = read_vis('twhya.vis.zarr')

# inspect some properties
print('times: ', mxds.xds0.time.values[0], ' to ', mxds.xds0.time.values[-1])
print('chans: ', mxds.xds0.chan.values[0], ' to ', mxds.xds0.chan.values[-1])
print('fields: ', mxds.fields.values)
```

```
overwrite_encoded_chunks True
times:  2012-11-19T07:37:00.000000000  to  2012-11-19T09:11:01.631999969
chans:  372533086425.9812  to  372766851074.4187
fields:  ['J0522-364' 'J0539+145' 'Ceres' 'J1037-295' 'TW Hya' 'TW Hya' '3c279']
```

We can directly select specific dimension indices if we know what we're looking for

[11]:
```
xds = mxds.xds0.isel(time=[76,77,78], chan=[6,7,8,12,20,21])

visplot(xds.DATA, ['time', 'chan'])
```



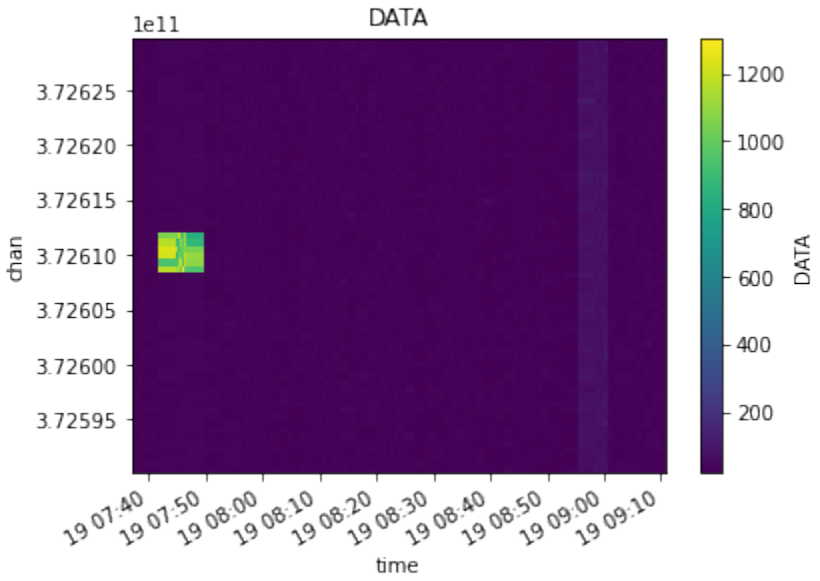
Or select by a range of dimension values, in this case lets select data between 372.59 and 372.63 GHz

[12]:
```
xds = mxds.xds0.sel(chan=slice(372.59e9, 372.63e9))
```

(continues on next page)

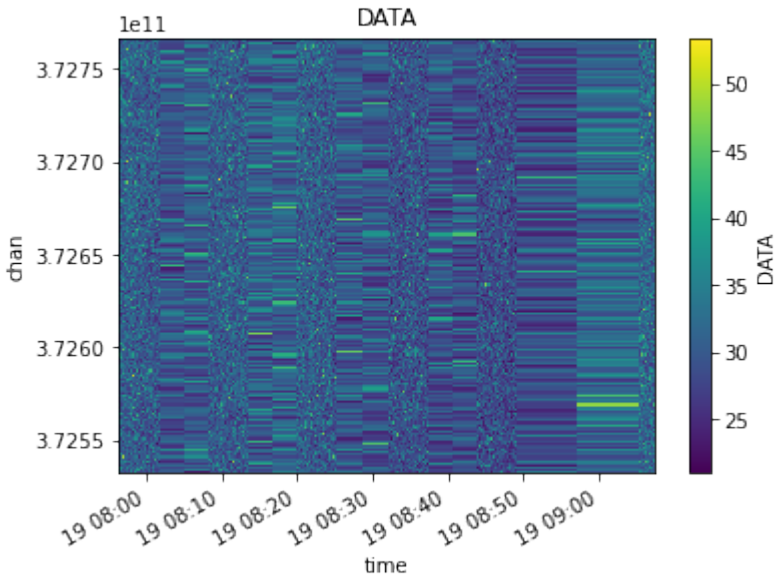
(continued from previous page)

```
visplot(xds.DATA, ['time', 'chan'])
```



Or select by a particular field value, in this case lets select just the TW Hya fields

[13]:
```
fields = mxds.field_ids[np.where(mxds.fields == 'TW Hya')].values

xds = mxds.xds0.where(mxds.xds0.FIELD_ID.isin(fields), drop=True)

visplot(xds.DATA, axis=['time','chan'])
```



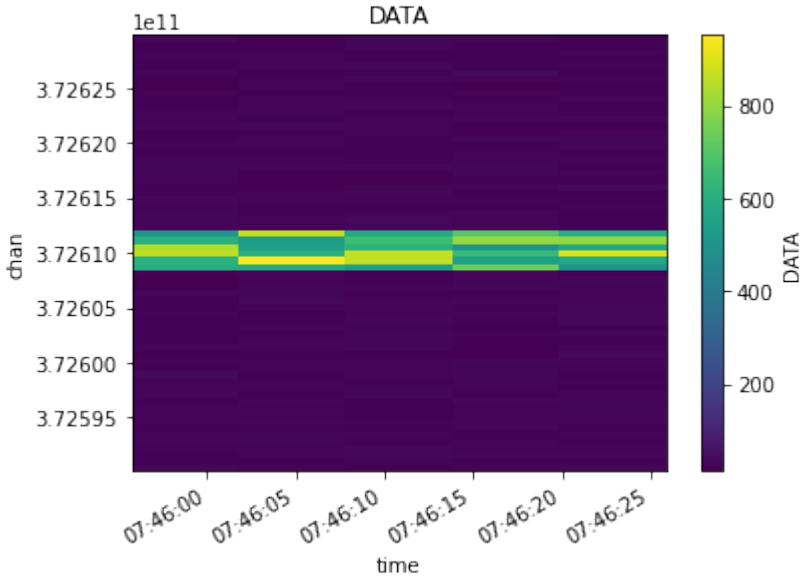
Finally, lets do a more complicated multi-selection

[14]:
```
xds = mxds.xds0.where(mxds.xds0.ANTENNA1.isin([2,3,4,5]) & mxds.xds0.ANTENNA2.isin([2,
↪3,4,5]) &
```

(continues on next page)

(continued from previous page)

```
                        (mxds.xds0.chan > 372.59e9) & (mxds.xds0.chan < 372.63e9) &
                        (mxds.xds0.time > np.datetime64('2012-11-19T07:45:50.0')) &
                        (mxds.xds0.time < np.datetime64('2012-11-19T07:46:30.0')),
                       drop=True)

visplot(xds.DATA, ['time', 'chan'])
```



Rather than cluttering each CNGI function with many parameters for data selection, instead the Dataset should be appropriately split before calling the relevant CNGI API function(s).

4.5 Splitting and Joining

Once you have the data selected that you want to work on, you can use split_dataset (was mstransform) to split that data out into its own mxds. The new mxds will include only the selected visibility and subtable data.

[15]:
```
from cngi.vis import split_dataset

# slice and split out the new mxds
xds_slice = mxds.xds0.sel(chan=slice(372.59e9, 372.63e9))
mxds_slice = mxds.assign_attrs(xds0=xds_slice) # replace xds0
mxds_slice = split_dataset(mxds_slice, 'xds0')

# we should now have a new mxds with less data, which we can see with a few print␣
↪statements
print("Select a subset of channels. Unrelated data (such as antennas) to the␣
↪remaining channels are dropped.")
print(f"old mxds: {len(mxds.xds0.chan)} chans, {len(mxds.antenna_ids)} antennas")
print(f"new mxds: {len(mxds_slice.xds0.chan)} chans, {len(mxds_slice.antenna_ids)}␣
↪antennas")
```

```
Select a subset of channels. Unrelated data (such as antennas) to the remaining␣
↪channels are dropped.
old mxds: 384 chans, 26 antennas
new mxds: 65 chans, 21 antennas
```

A split mxds, or several mxds gathered from different telescopes are times, can be merged back into a single mxds for further calibration or computation with join_dataset (was concat).

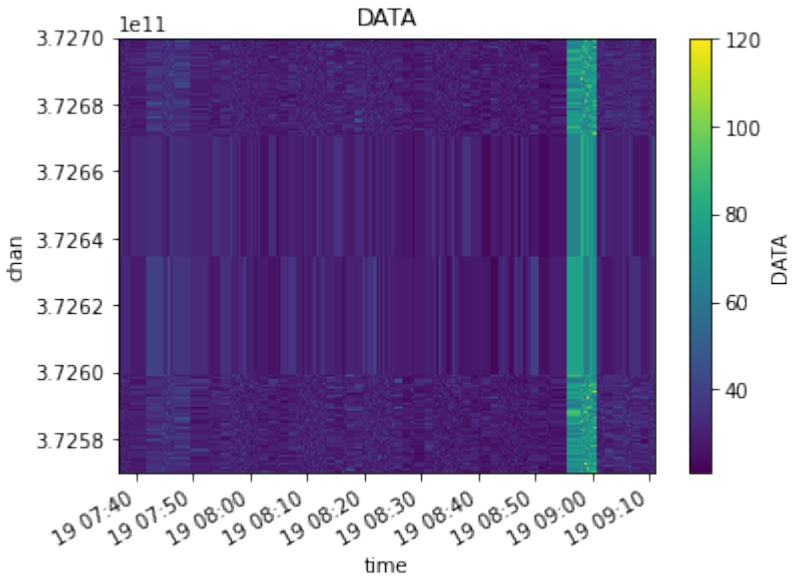
[16]:
```
from cngi.dio import read_vis
from cngi.vis import join_dataset
import numpy as np

# open the master xds
mxds = read_vis('twhya.vis.zarr')

# split out several channels to run computations on
mxds_slice0 = mxds.assign_attrs(xds0=mxds.xds0.sel(chan=slice(372.57e9, 372.60e9)))
mxds_slice1 = mxds.assign_attrs(xds0=mxds.xds0.sel(chan=slice(372.67e9, 372.70e9)))
# do channel-dependent operations...

# merge the split mxds into a single mxds (takes some time)
tmp_mxds = join_dataset(mxds_slice0, mxds_slice1) # mxds_slice1.xds0 is moved to mxds_
↪cal.xds1
list(filter(lambda n: 'xds' in n, tmp_mxds.attrs.keys()))
```

```
overwrite_encoded_chunks True
Warning: reference value -1 in subtable WEATHER does not exist in ANTENNA.antenna_id!
Warning: reference value 5 in subtable FIELD does not exist in SOURCE.source_id!
```

[16]:
```
['xds0', 'xds1']
```

Similarly, visibilities can be merged with join_vis. Think really hard about when it is appropriate to use this function, however, since some meaning inherent to the data could be lost by doing further computation on the merged dataset.

[17]:
```
from cngi.vis import join_vis
from cngi.vis import visplot

# merge the slices back together
mxds_joined = join_vis(tmp_mxds, 'xds0', 'xds1')
visplot(mxds_joined.xds0.DATA, ['time', 'chan'])

# compare to the extra data in channels 372.60-372.62 which outshines everything
# xds_compare = mxds.xds0.sel(chan=slice(372.57e9, 372.70e9))
# visplot(xds_compare.DATA, ['time', 'chan'])
```



4.6 Flagging

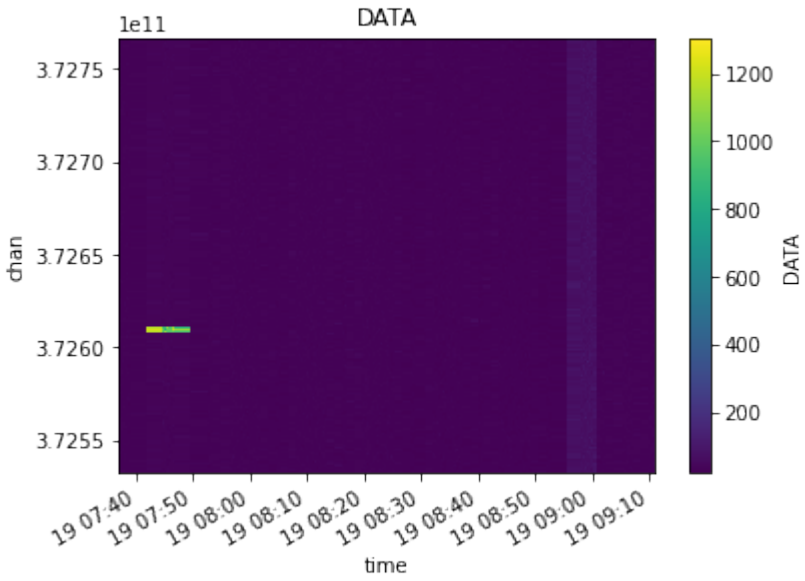
Any boolean data variable can be used to flag any other data variable of common dimension(s).

Flagged values are set to np.nan, and subsequent math/analysis should be of the type that ignores nan values.

First lets see the original raw data

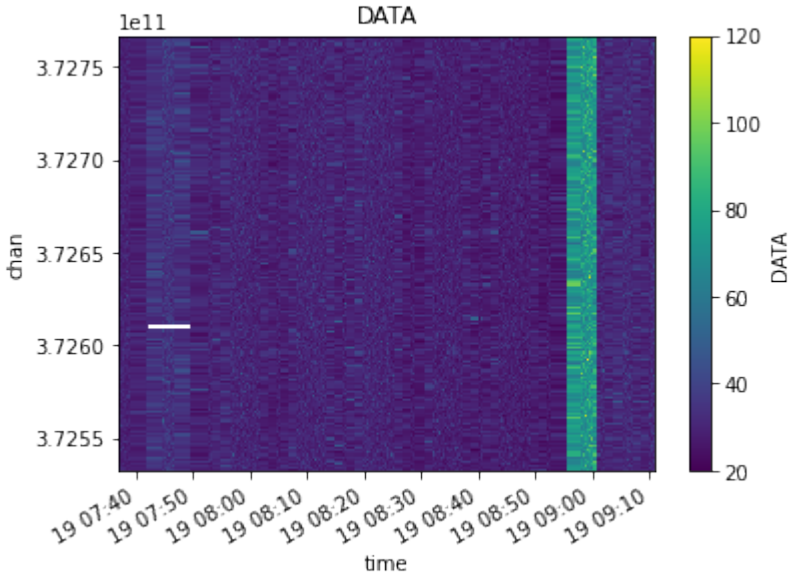
[18]:
```
from cngi.dio import read_vis

mxds = read_vis('twhya.vis.zarr')

visplot(mxds.xds0.DATA, ['time', 'chan'])
```

```
overwrite_encoded_chunks True
```



Now lets flag the entire dataset contents based on the value of the FLAG data variable. The bright line from Ceres is replaced with Nan's (that appear as blank whitespace in the plot)

[19]:
```
from cngi.vis import apply_flags

flagged_xds = apply_flags(mxds, 'xds0', flags='FLAG')

visplot(flagged_xds.xds0.DATA, ['time', 'chan'])
```



4.7 Averaging and Smoothing

Averaging functions will change the shape of the resulting xarray dataset along the dimension being averaged.

Smoothing functions always return an xarray dataset with the same dimensions as the original.

We will use the TWHya dataset for this section. Dask may emit performance warnings if the chunk size is too small. We can increase our chunk size in the xds to any multiple of the chunk size used during conversion. We will do that here to avoid the performance warnings for this section. Note that baseline is already chunked at the maximum for its dimension size, so we can omit it.

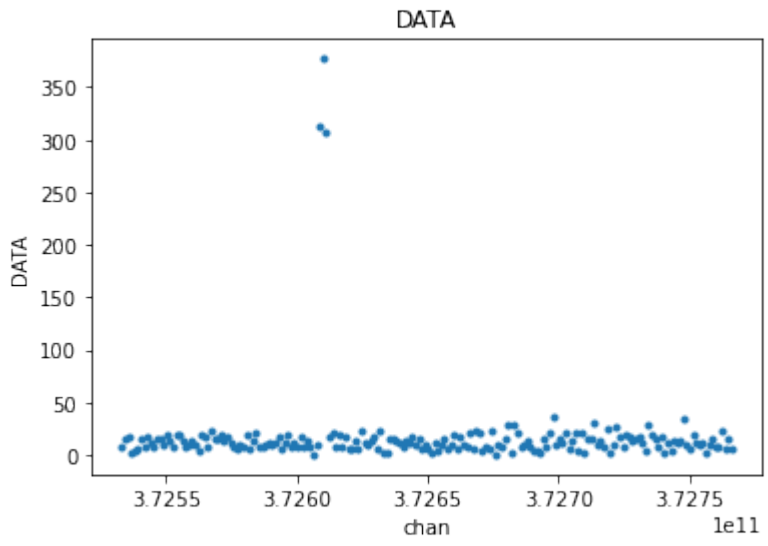
[20]:
```
from cngi.dio import read_vis

mxds = read_vis('twhya.vis.zarr', chunks={'time':200, 'chan':64, 'pol':2})
```

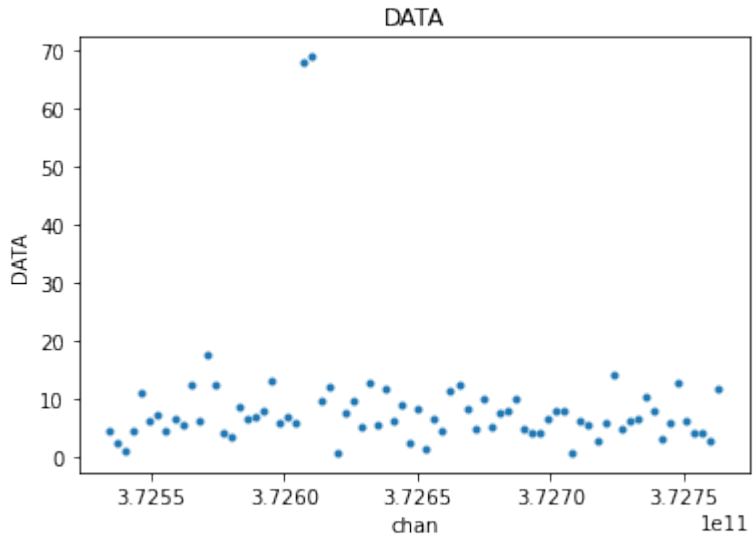
```
overwrite_encoded_chunks True
```

4.7.1 Channel Averaging

The channel averaging function looks for all data variables in the dataset with a channel dimension and averages by the specified bin width. The returned dataset will have a different channel dimension size.

[21]:
```
from cngi.vis import chan_average, visplot

# average 5 channels of original unflagged data
avg_xds = chan_average(mxds, 'xds0', width=5)

# compare the original to the channel averaged
visplot(mxds.xds0.DATA[30,0,:,0], 'chan')
visplot(avg_xds.xds0.DATA[30,0,:,0], 'chan')
```





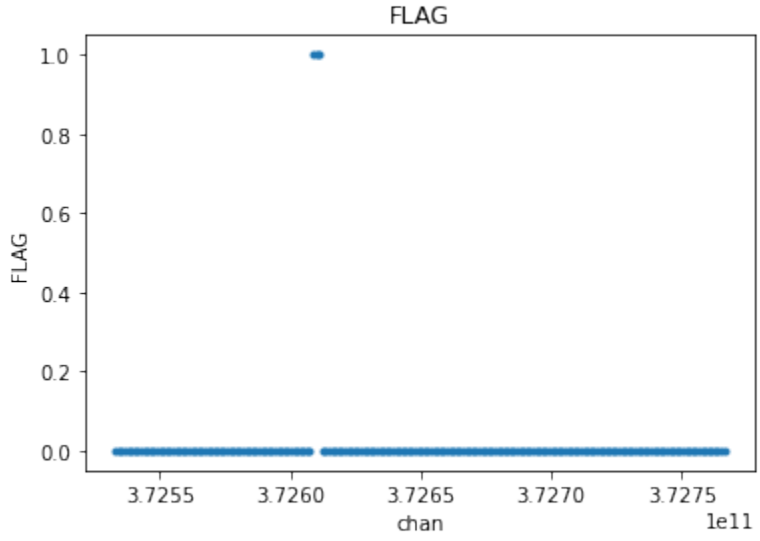
[22]:
```
# confirm new channel dimension after averaging
print(dict(mxds.xds0.dims))
print(dict(avg_xds.xds0.dims))
```

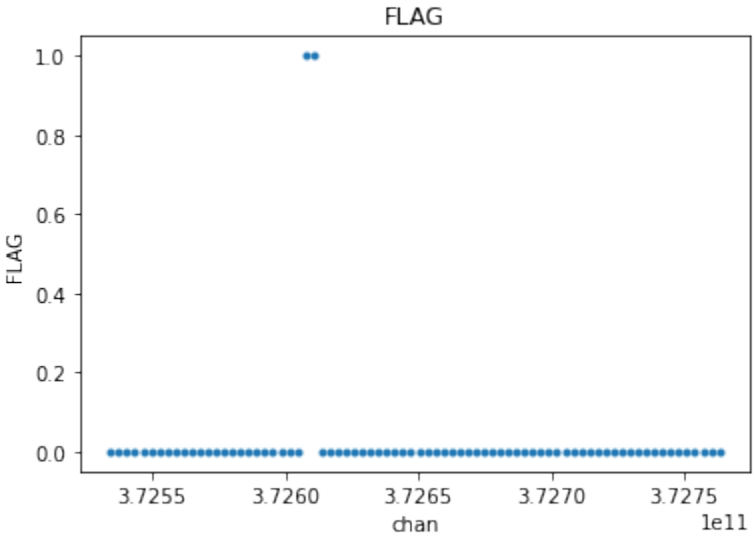
```
{'baseline': 210, 'chan': 384, 'pol': 2, 'pol_id': 1, 'spw_id': 1, 'time': 410, 'uvw_
↪index': 3}
{'baseline': 210, 'chan': 76, 'pol': 2, 'pol_id': 1, 'spw_id': 1, 'time': 410, 'uvw_
↪index': 3}
```

Since every variable with a channel dimension in the dataset is averaged, this will also include the FLAG variable. FLAG is a boolean type with values of 0 or 1 that are averaged over the width, resulting in a decimal number. The result is then typcast back to boolean, which is the same as just rounding up.

Long story short, if any channel in the width is flagged, the resulting averaged channel will also be flagged

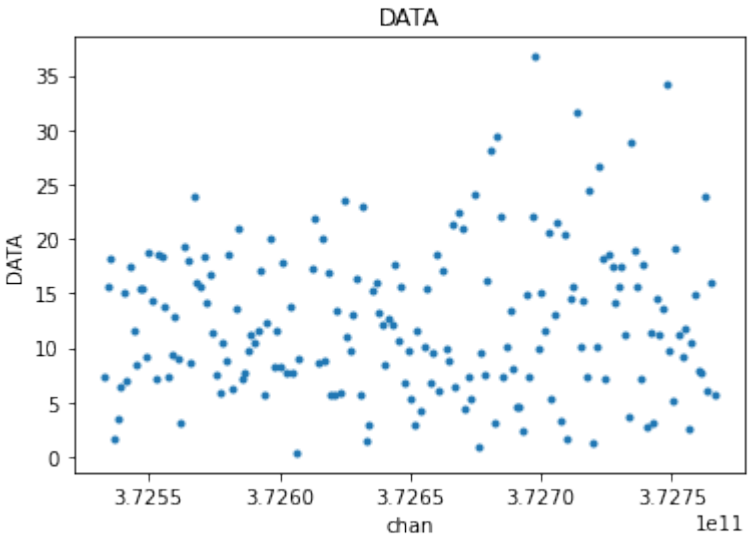
[23]:
```
# compare the original flags to the channel averaged flags
visplot(mxds.xds0.FLAG[30,0,:,0], 'chan')
visplot(avg_xds.xds0.FLAG[30,0,:,0], 'chan')
```

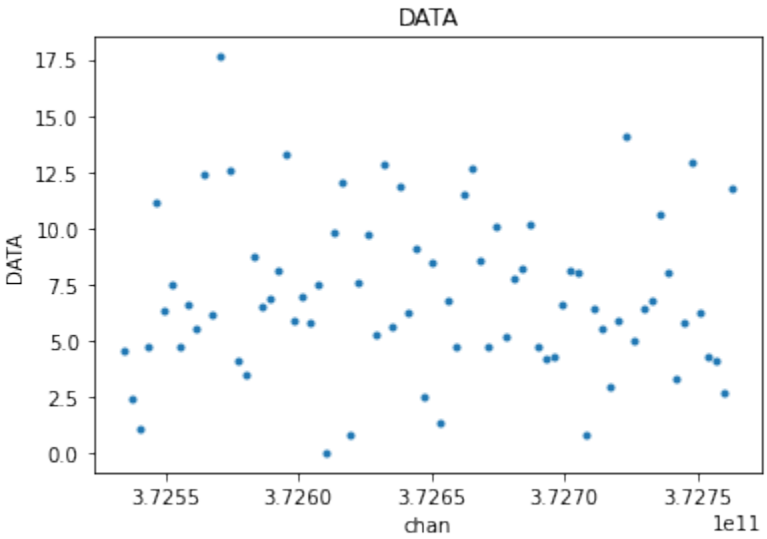




One might want to apply flags before channel averaging

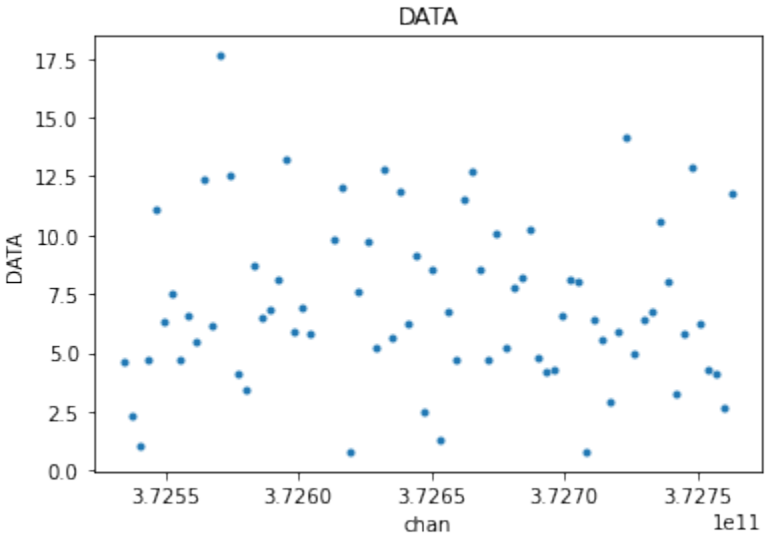
[24]:
```
from cngi.vis import apply_flags

flg_xds = apply_flags(mxds, 'xds0', flags=['FLAG'])
avg_flg_xds = chan_average(flg_xds, 'xds0', width=5)

visplot(flg_xds.xds0.DATA[30,0,:,0], 'chan')
visplot(avg_flg_xds.xds0.DATA[30,0,:,0], 'chan')
```





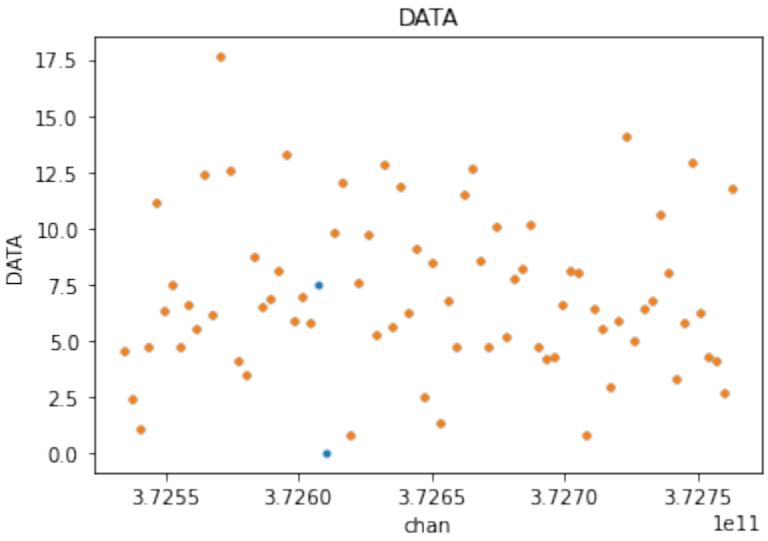
Or apply flags after channel averaging

[25]:
```
flg_avg_xds = apply_flags(avg_xds, 'xds0', flags=['FLAG'])

visplot(flg_avg_xds.xds0.DATA[30,0,:,0], 'chan')
```



There is a small difference of two points, made more clear when we overlay the plots

[26]:
```
visplot(avg_flg_xds.xds0.DATA[30,0,:,0], 'chan', drawplot=False)
visplot(flg_avg_xds.xds0.DATA[30,0,:,0], 'chan', overplot=True)
```

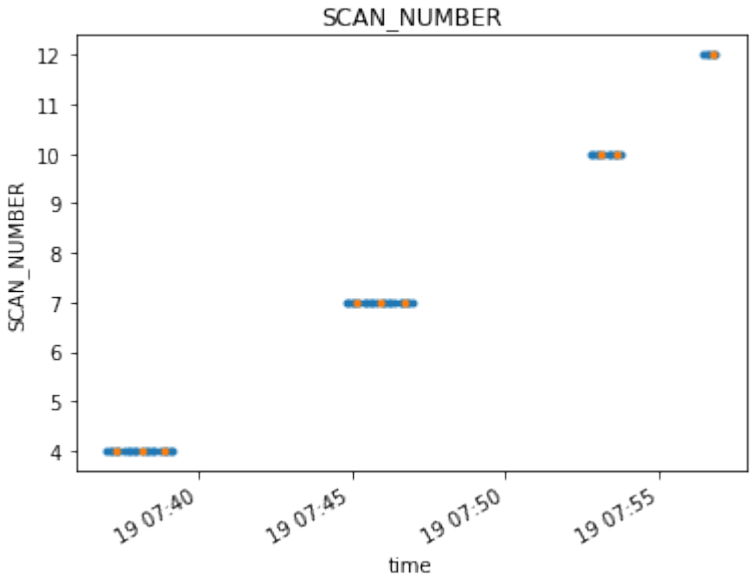


4.7.2 Time Averaging

The time averaging function can combine adjacent time steps or resample to a new uniform time step. This affects all fields in the xds that have a time dimension. The span parameter can be used to limit averaging to just the time steps within each scan, or within each state. The returned dataset will have a different time dimension size.

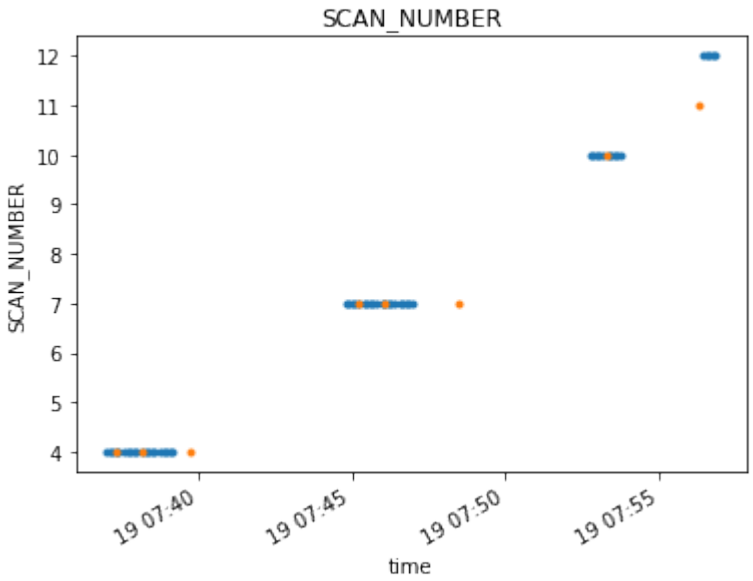
First let's average across the states in a scan using a bin of 7

[27]:
```
from cngi.vis import time_average

state_xds = time_average(mxds,'xds0', bin=7, span='state')

visplot(mxds.xds0.SCAN_NUMBER.where(mxds.xds0.time < np.datetime64('2012-11-19T07:57:
↪00.0'), drop=True), 'time', drawplot=False)
visplot(state_xds.xds0.SCAN_NUMBER.where(state_xds.xds0.time < np.datetime64('2012-11-
↪19T07:57:00.0'), drop=True), 'time', overplot=True)
```



Averaging across both state and scan allows the bins to cross the time gaps between scans.

[28]:
```
both_xds = time_average(mxds, 'xds0', bin=7, span='both')

visplot(mxds.xds0.SCAN_NUMBER.where(mxds.xds0.time < np.datetime64('2012-11-19T07:57:
↪00.0'), drop=True), 'time', drawplot=False)
visplot(both_xds.xds0.SCAN_NUMBER.where(both_xds.xds0.time < np.datetime64('2012-11-
↪19T07:57:00.0'), drop=True), 'time', overplot=True)
```



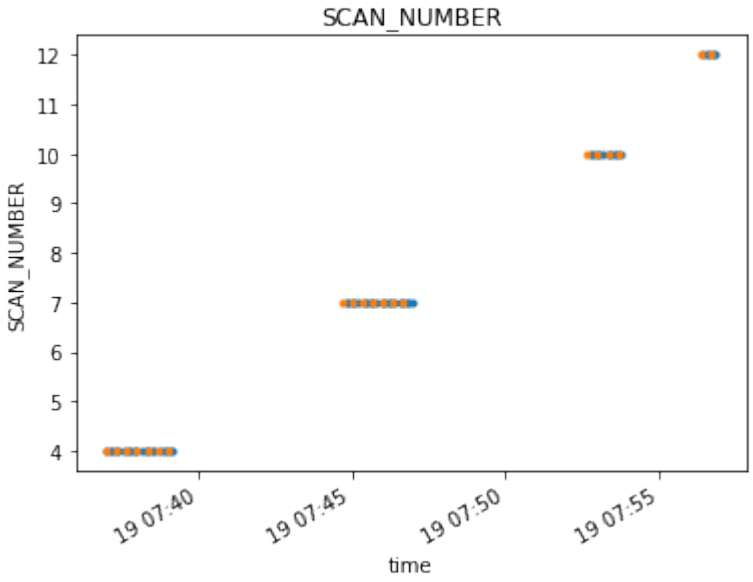
Switching to the width parameter will resample the contents of the xds to a uniform time bin across the span

[29]:
```
resampled_xds = time_average(mxds, 'xds0', width='20s', span='state')

visplot(mxds.xds0.SCAN_NUMBER.where(mxds.xds0.time < np.datetime64('2012-11-19T07:57:
↪00.0'), drop=True), 'time', drawplot=False)
```

(continues on next page)

(continued from previous page)

```
visplot(resampled_xds.xds0.SCAN_NUMBER.where(resampled_xds.xds0.time < np.datetime64(
↪'2012-11-19T07:57:00.0'), drop=True), 'time', overplot=True)
```



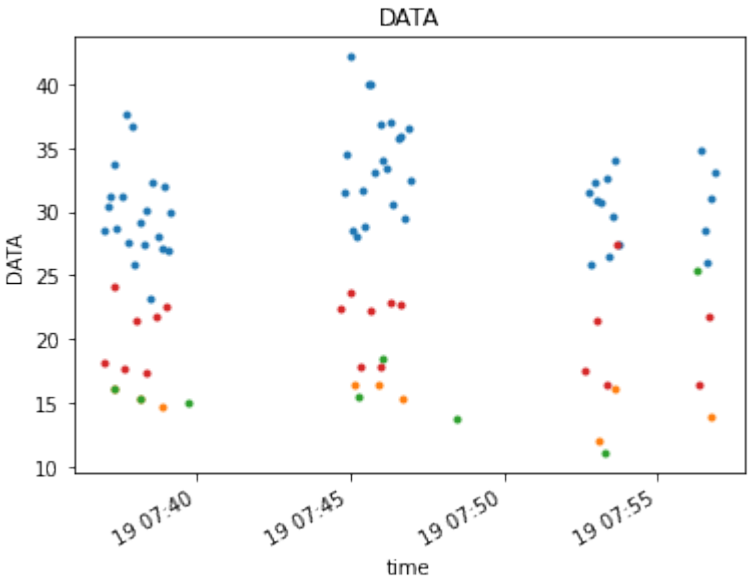
In all cases, the dimensionality of the time averaged xds returned will be different than the start.

[30]:
```
print(dict(mxds.xds0.dims))
print(dict(state_xds.xds0.dims))
print(dict(both_xds.xds0.dims))
print(dict(resampled_xds.xds0.dims))
```

```
{'baseline': 210, 'chan': 384, 'pol': 2, 'pol_id': 1, 'spw_id': 1, 'time': 410, 'uvw_
↪index': 3}
{'baseline': 210, 'chan': 384, 'pol': 2, 'pol_id': 1, 'spw_id': 1, 'time': 68, 'uvw_
↪index': 3}
{'baseline': 210, 'chan': 384, 'pol': 2, 'pol_id': 1, 'spw_id': 1, 'time': 59, 'uvw_
↪index': 3}
{'baseline': 210, 'chan': 384, 'pol': 2, 'pol_id': 1, 'spw_id': 1, 'time': 150, 'uvw_
↪index': 3}
```

We can inspect the affect of different time averaging methods on the DATA contents

[31]:
```
timefilter = np.datetime64('2012-11-19T07:57:00.0')
visplot(mxds.xds0.DATA.where(mxds.xds0.time < timefilter, drop=True)[:,:,100,0], 'time
↪', drawplot=False)
visplot(state_xds.xds0.DATA.where(state_xds.xds0.time < timefilter, drop=True)[:,:,
↪100,0], 'time', overplot=True, drawplot=False)
visplot(both_xds.xds0.DATA.where(both_xds.xds0.time < timefilter, drop=True)[:,:,100,
↪0], 'time', overplot=True, drawplot=False)
visplot(resampled_xds.xds0.DATA.where(resampled_xds.xds0.time < timefilter,␣
↪drop=True)[:,:,100,0], 'time', overplot=True)
```

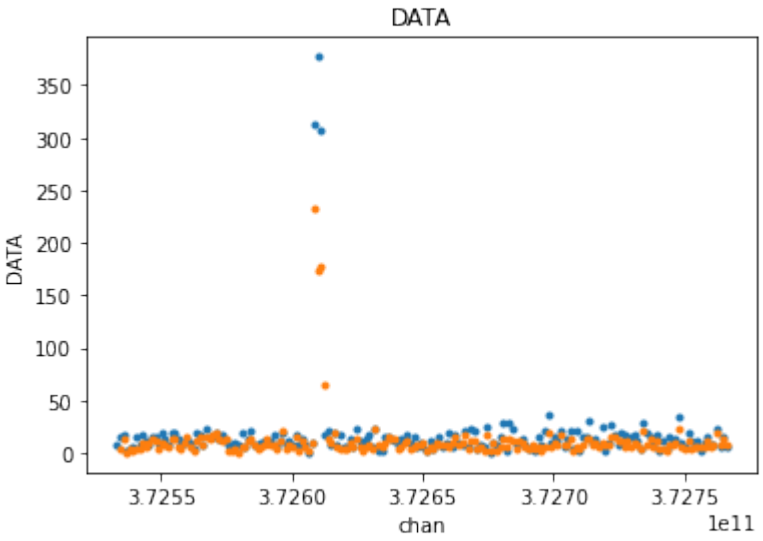


Flagging works the same way as in channel averaging. Flags are averaged with all other data sharing the time axis. They may be applied before or after time averaging. The time averaged flag field is converted back to boolean. Any single flagged value in the original data will cause the entire bin to be flagged in the resulting data.

4.7.3 Channel Smoothing

Channel smoothing allows a variety of window shapes to be convolved across the channel dimension to smooth over changes from bin to bin. The standard CASA hanning smooth is supported as the default option. The returned dataset will have the same dimensions as the original.

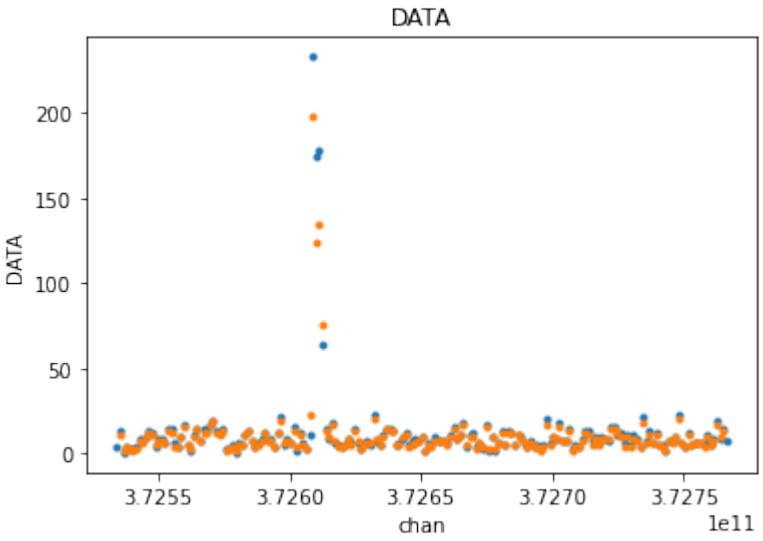
[32]:
```
from cngi.vis import chan_smooth

smooth_xds = chan_smooth(mxds, 'xds0')

visplot(mxds.xds0.DATA[30,0,:,0], 'chan', drawplot=False)
visplot(smooth_xds.xds0.DATA[30,0,:,0], 'chan', overplot=True)
```



chansmooth uses the scipy.signal package to set the window shape. Therefore, all the window functions supported by scipy are also supported here. The size of the window width is configurable.

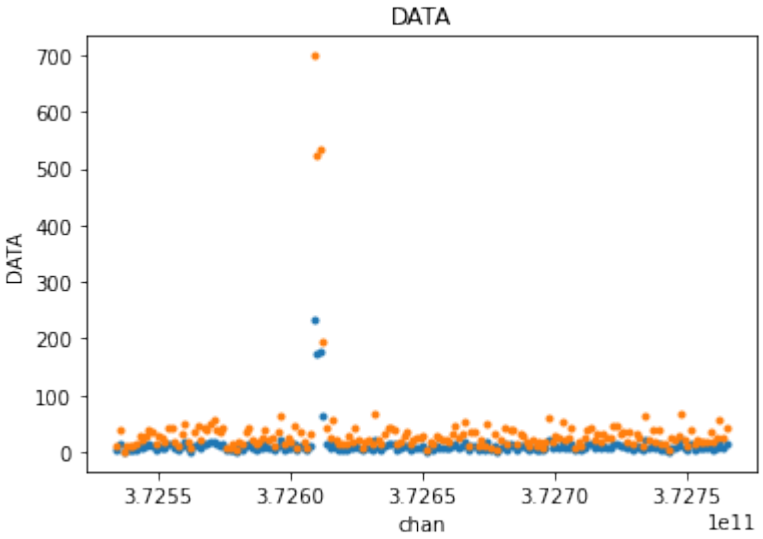
https://docs.scipy.org/doc/scipy/reference/signal.windows.html#module-scipy.signal.windows

[33]:
```
bohman_xds = chan_smooth(mxds, 'xds0', type='bohman', size=7)

visplot(smooth_xds.xds0.DATA[30,0,:,0], 'chan', drawplot=False)
visplot(bohman_xds.xds0.DATA[30,0,:,0], 'chan', overplot=True)
```



non-unity gains are also supported to amplify or attenuate the output

[34]:
```
hann_xds = chan_smooth(mxds, 'xds0', type='hann', size=5, gain=1.0)
amp_hann_xds = chan_smooth(mxds, 'xds0', type='hann', size=5, gain=3.0)

visplot(hann_xds.xds0.DATA[30,0,:,0], 'chan', drawplot=False)
visplot(amp_hann_xds.xds0.DATA[30,0,:,0], 'chan', overplot=True)
```



4.8 UV Fitting

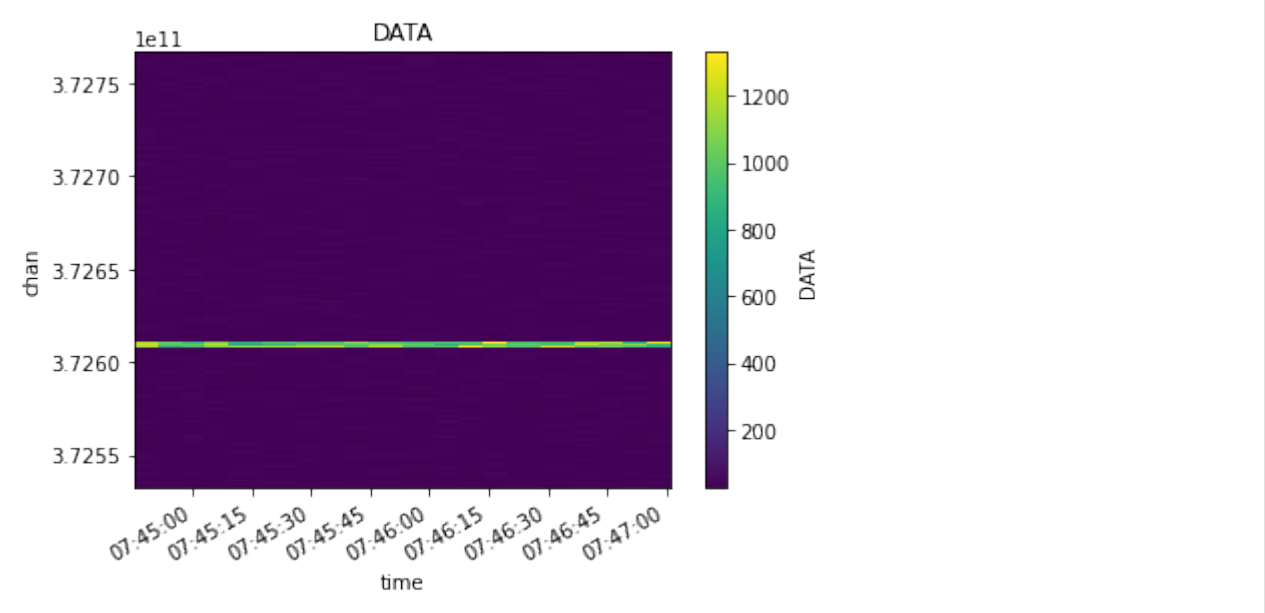
Various modeling can be performed in the UV-domain through polynomial regression on the visibility data.

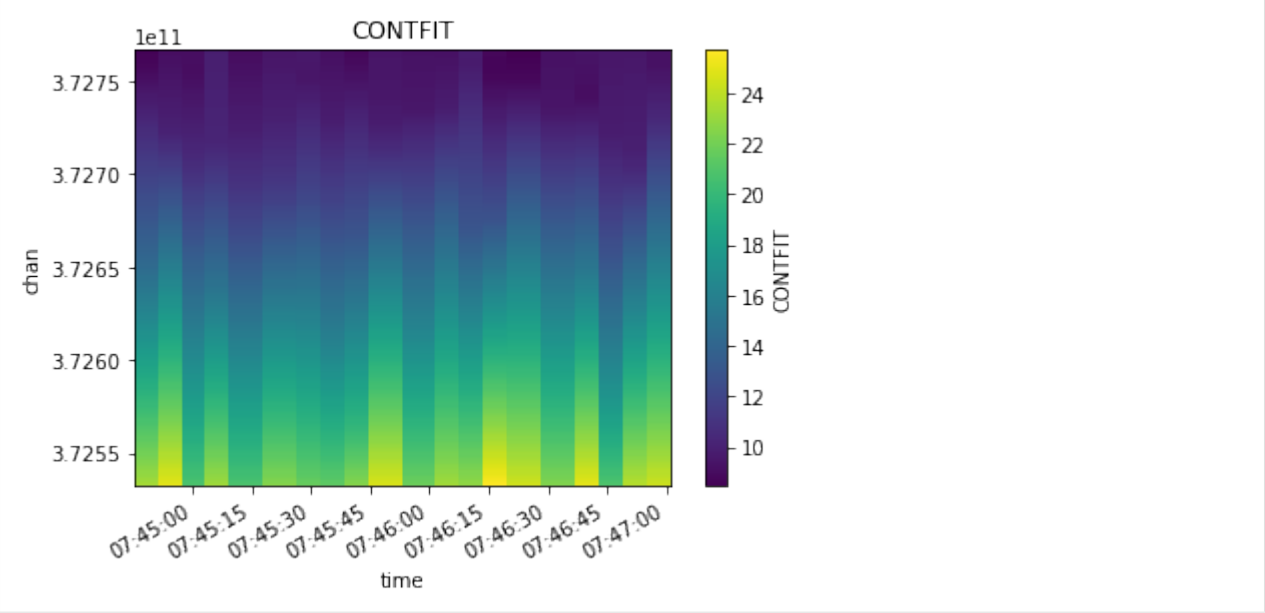
The uvcontfit function takes a source data variable in the xarray dataset and fits a polynomial of fitorder across the channel axis. The resulting model is placed in the target data variable and retains the dimensionality of the source. Specified channels may be excluded to focus on fitting only continuum visibilities and not line emissions.

In our example dataset we know that the Ceres has a line emission, so let's focus on just that field. We will need to master xarray dataset to see which field ID(s) is Ceres.

[35]:
```
from cngi.dio import read_vis
from cngi.vis import uv_cont_fit, visplot
import dask.config
dask.config.set({"array.slicing.split_large_chunks": False})

mxds = read_vis('twhya.vis.zarr')

fields = mxds.field_ids[np.where(mxds.fields == 'Ceres')].values

mxds = mxds.assign_attrs({'ceres_xds':mxds.xds0.where(mxds.xds0.FIELD_ID.isin(fields),
↪ drop=True)})

fit_xds = uv_cont_fit(mxds, 'ceres_xds', source='DATA', target='CONTFIT')

visplot(fit_xds.ceres_xds.DATA, ['time', 'chan'])
visplot(fit_xds.ceres_xds.CONTFIT, ['time', 'chan'])
```

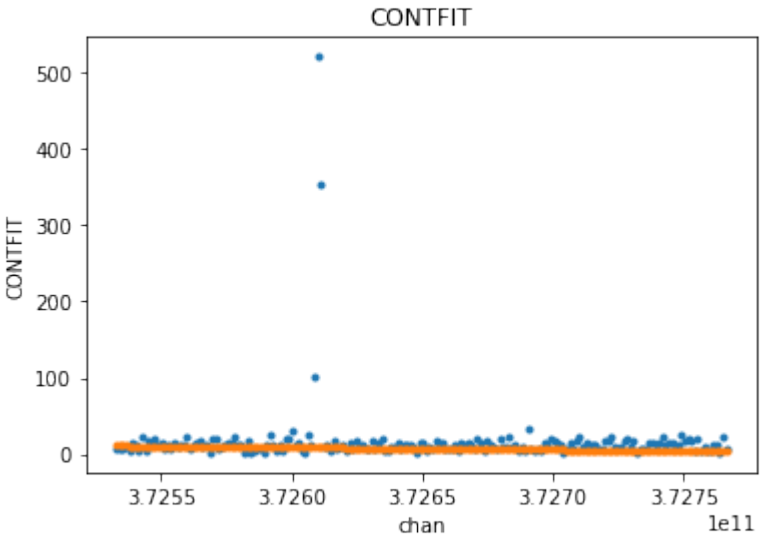
```
overwrite_encoded_chunks True
```





What we are seeing is merged across baselines and polarizations. Let's look at just one time/baseline/pol sample and examine the fit. The linear fit is skewed by the line emission towards one side of the channel axis.

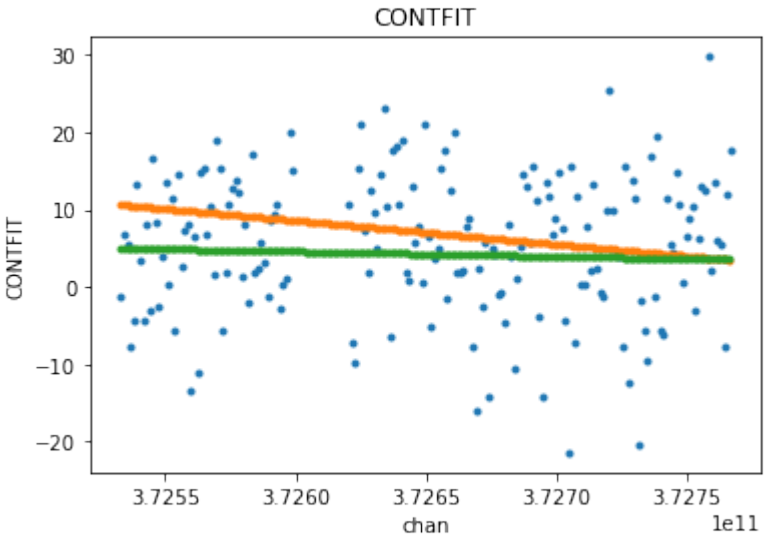
[36]:
```
visplot(fit_xds.ceres_xds.DATA[10,10,:,0], ['chan'], drawplot=False)
visplot(fit_xds.ceres_xds.CONTFIT[10,10,:,0], ['chan'], overplot=True)
```



Let's omit the channels with the line emission and do the fit again. We should see less skewing in the fitted continuum.

The fits appear lower in value than the mean of the data when viewing the magnitude of complex visibilities. The real and imaginary parts are fitted individually. We examine just the real piece here to see a that the fit is indeed a decent approximation through the center of the data.

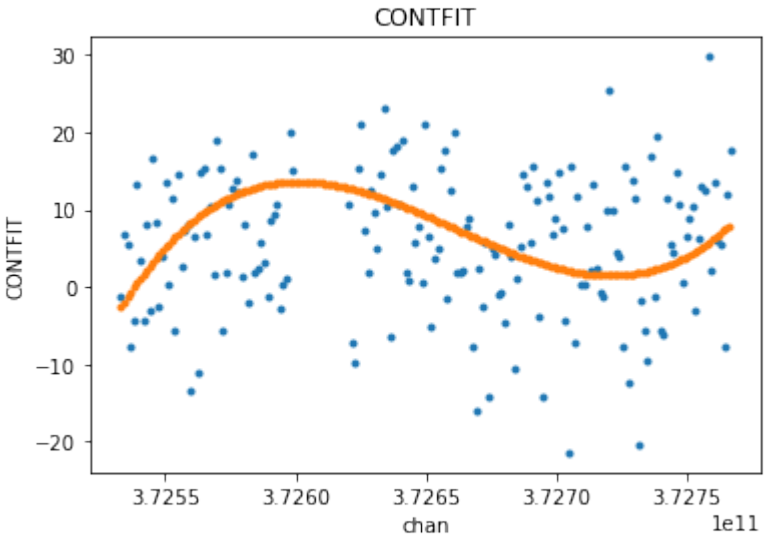
[37]:
```
excludechans = np.where( (mxds.xds0.chan > 3.726e11) & (mxds.xds0.chan < 3.7262e11)␣
↪)[0]

fit_xds2 = uv_cont_fit(mxds, 'ceres_xds', source='DATA', target='CONTFIT', fitorder=1,
↪ excludechans=excludechans)

includechans = np.setdiff1d( range(mxds.xds0.dims['chan']), excludechans)
visplot(fit_xds2.ceres_xds.DATA[10,10,includechans,0].real, ['chan'], drawplot=False)
visplot(fit_xds.ceres_xds.CONTFIT[10,10,:,0].real, ['chan'], drawplot=False,␣
↪overplot=True)
visplot(fit_xds2.ceres_xds.CONTFIT[10,10,:,0].real, ['chan'], overplot=True)
```



For even more fun, we can try a higher order fit. Let's use all channels to allow the line emission to exaggerate the

non-linearity, but only plot the continuum DATA channels so the y-scale stays smaller.

[38]:
```
fit_xds3 = uv_cont_fit(mxds, 'ceres_xds', source='DATA', target='CONTFIT', fitorder=3,
↪ excludechans=[])

visplot(fit_xds3.ceres_xds.DATA[10,10,includechans,0].real, ['chan'], drawplot=False)
visplot(fit_xds3.ceres_xds.CONTFIT[10,10,:,0].real, ['chan'], overplot=True)
```



Several metrics related to the quality of the fit are stored in the xarray dataset attributes section as metadata. They are named with a prefix matching the target parameter in uvcontfit.

[39]:
```
metrics = [kk for kk in fit_xds.ceres_xds.attrs.keys() if kk.startswith('CONTFIT')]
print(metrics)

print(fit_xds.ceres_xds.CONTFIT_rms_error)
```

```
['CONTFIT_rms_error', 'CONTFIT_min_max_error', 'CONTFIT_bw_frac', 'CONTFIT_freq_frac']
<xarray.DataArray ()>
dask.array<pow, shape=(), dtype=complex128, chunksize=(), chunktype=numpy.ndarray>
```

One thing to note is that these metrics are dask elements, and are not actually computed until explicity requested. Things like visplot explicity force the computation of the data needed for plotting, but nothing has forced these metrics to be computed yet. So we will call .compute() to see their values.

[40]:
```
print('rms error with line included: ', fit_xds.ceres_xds.CONTFIT_rms_error.values)
print('rms error with line excluded: ', fit_xds2.ceres_xds.CONTFIT_rms_error.values)
```

```
rms error with line included:  (16.444908051664182-0.7566763926282181j)
rms error with line excluded:  (0.37306136672597817-0.007993456030923675j)
```

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/images.ipynb

CHAPTER

FIVE

IMAGES

An overview of the image data structure and manipulation.

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here

[1]:
```
# Installation
# for this demonstration we will use the data from the ALMA First Look at Imaging␣
↪CASAguide
import os, warnings
warnings.simplefilter("ignore", category=RuntimeWarning)  # suppress warnings about␣
↪nan-slices
print("installing casa6 and cngi (takes a few minutes)...")
os.system("apt-get install libgfortran3")
os.system("pip install casatasks==6.3.0.48")
os.system("pip install casadata")
os.system("pip install cngi-prototype==0.0.92")

print("downloading MeasurementSet from CASAguide First Look at Imaging...")
!gdown -q --id 1N9QSs2Hbhi-BrEHx5PA54WigXt8GGgx1
!tar -xzf sis14_twhya_calibrated_flagged.ms.tar.gz

print('complete')
```

```
installing casa6 and cngi (takes a few minutes)...
downloading MeasurementSet from CASAguide First Look at Imaging...
complete
```

5.1 Initialize the Environment

`InitializeFramework` instantiates a client object (does not need to be returned and saved by caller). Once this object exists, all Dask objects automatically know to use it for parallel execution.

```
>>> from cngi.direct import InitializeFramework
>>> client = InitializeFramework(workers=4, memory='2GB')
>>> print(client)
<Client: 'tcp://127.0.0.1:33227' processes=4 threads=4, memory=8.00 GB>
```

Omitting this step will cause the subsequent Dask dataframe operations to use the default built-in scheduler for parallel execution (which can actually be faster on local machines anyway)

Google Colab doesn't really support the dask.distributed environment particularly well, so we will let Dask use its default scheduler.

5.2 Create Image Data

First we need to create a CASA image by calling CASA6 tclean on the CASAguide MS

Pro tip: you can see the files being created by expanding the left navigation bar in colab (little arrow on top left) and going to "Files"

[2]:
```
from casatasks import tclean
tclean(vis='sis14_twhya_calibrated_flagged.ms', imagename='sis14_twhya_calibrated_
↪flagged', field='5', spw='',
       specmode='cube', deconvolver='hogbom', nterms=1, imsize=[250,250], gridder=
↪'standard', cell=['0.1arcsec'],
       nchan=10, weighting='natural', threshold='0mJy', niter=5000, interactive=False,
↪ pbcor=True,
       savemodel='modelcolumn', usemask='auto-multithresh')
print('complete')
```

```
complete
```

tclean produces an image along with a number of supporting products (residual, pb, psf, etc) in their own separate directories.

CNGI uses an xarray data model and the zarr storage format which is capable of storing all image products (of the same shape) together.

Note that the .mask image product will be renamed to 'automask' in the xarray data model and 'mask' will hold the mask pixel values extracted from the .image.

[3]:
```
from cngi.conversion import convert_image

image_xds = convert_image('sis14_twhya_calibrated_flagged.image')
```

```
converting Image...
processed image in 1.7558475 seconds
```

Within the xarray dataset image, we can see a very clear definition of the image properties. The **Dimensions** section holds the size of the image, while the **Coordinates** section defines the values of each dimension. The actual image (and supporting products) are stored in the **Data variables** section. Lastly, the **Attributes** section holds the remaining metadata.

Note that the variables *image*, *automask*, *mask*, *model*, *pb*, *psf*, and *residual* all share the same dimensions of 250x250x1x10 while the variable *sumwt* is a subset of dimensions 1x10

[4]:
```
print(image_xds)
```

```
<xarray.Dataset>
Dimensions:             (chan: 10, l: 250, m: 250, pol: 1, time: 1)
Coordinates:
  * chan                (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11
    declination         (l, m) float64 dask.array<chunksize=(250, 250), meta=np.
↪ndarray>
  * l                   (l) float64 6.06e-05 6.012e-05 ... -5.963e-05 -6.012e-05
  * m                   (m) float64 -6.06e-05 -6.012e-05 ... 5.963e-05 6.012e-05
  * pol                 (pol) float64 1.0
    right_ascension     (l, m) float64 dask.array<chunksize=(250, 250), meta=np.
↪ndarray>
  * time                (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    AUTOMASK            (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    PB                 (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0           (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes: (12/19)
    axisnames:             ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:             ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:            [0.6526992917060852, 0.5043594837188721, -65.889526...
    commonbeam_units:      ['arcsec', 'arcsec', 'deg']
    date_observation:      2012/11/19/07
    direction_reference:   j2000
    ...                    ...
    restoringbeam:         [0.6526992917060852, 0.5043594837188721, -65.889526...
    spectral__reference:   lsrk
    telescope:             alma
    telescope_position:    [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                  Jy/beam
    velocity__type:        radio
```

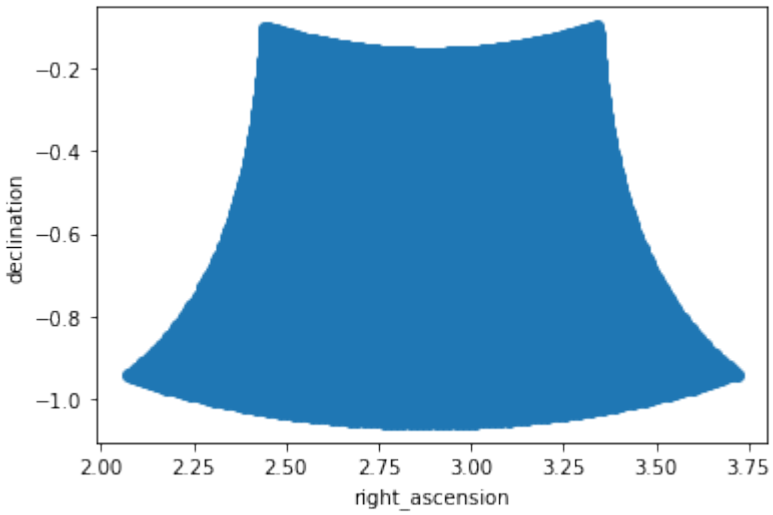
The image has both cartesian pixel coordinates (l, m) and spherical world coordinates (right_ascension, declination).

The spherical nature of the multi-dimensional right_ascension and declination coordinate structures becomes clearer when we plot a very wide image:

[5]:
```
tclean(vis='sis14_twhya_calibrated_flagged.ms', imagename='wide_cube', field='5', spw=
↪'', specmode='cube', cell='30arcmin',
       imsize=100, nchan=10, deconvolver='hogbom', niter=10, savemodel='modelcolumn',␣
↪usemask='auto-multithresh')

widexds = convert_image('wide_cube.image')
widexds.plot.scatter(x='right_ascension', y='declination')
```

```
converting Image...
processed image in 0.68170214 seconds
```

[5]:
```
<matplotlib.collections.PathCollection at 0x7fdc64b238d0>
```

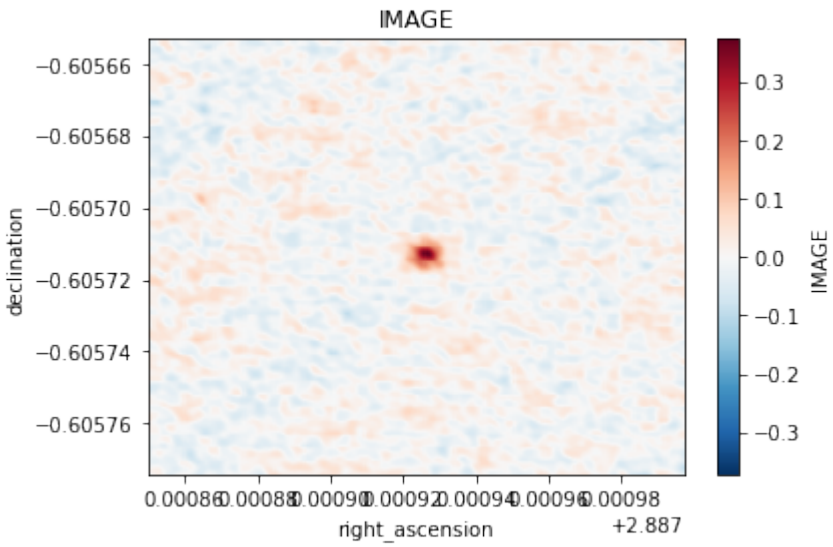


5.3 Preview Image

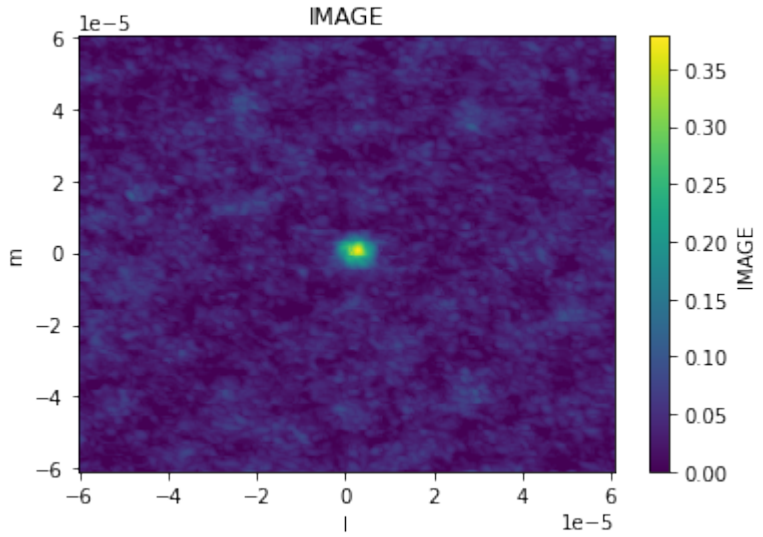
We can quickly spot check image contents. This is also handy later on during image manipulation and analysis. Most image data is four-dimensional, so the plotting routine will collapse extra dimensions by taking the max (this is more robust than mean).

The image can be plotted using its spherical world coordinates or cartesian pixel indices

[6]:
```
from cngi.image import implot

# just channel 5
implot(image_xds.IMAGE, axis=['right_ascension', 'declination'], chans=5)
```

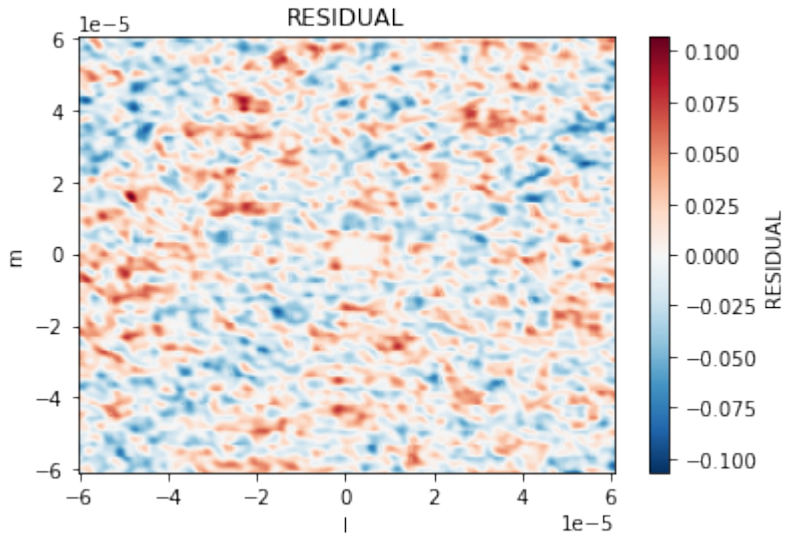


[7]:
```
# max of all channels, with cartesian pixel indices
implot(image_xds.IMAGE, axis=['l', 'm'], chans=None)
```



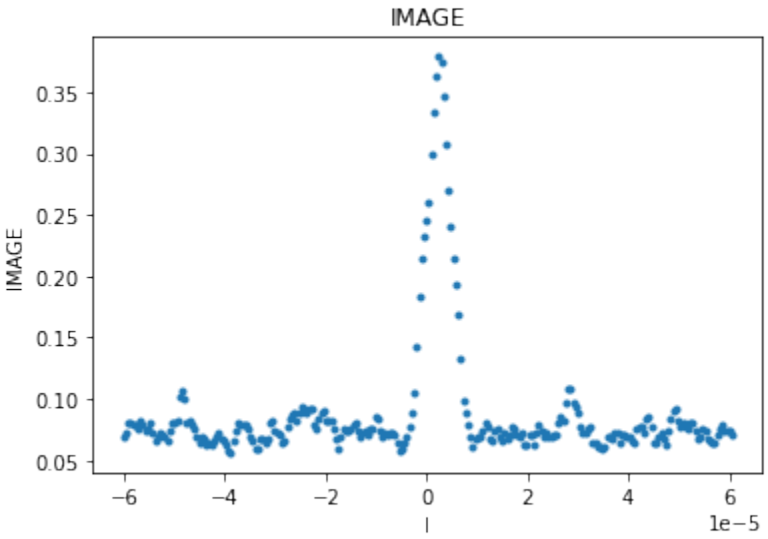
Other image data products are plotted the same way

[8]:
```
implot(image_xds.RESIDUAL, chans=5)
```



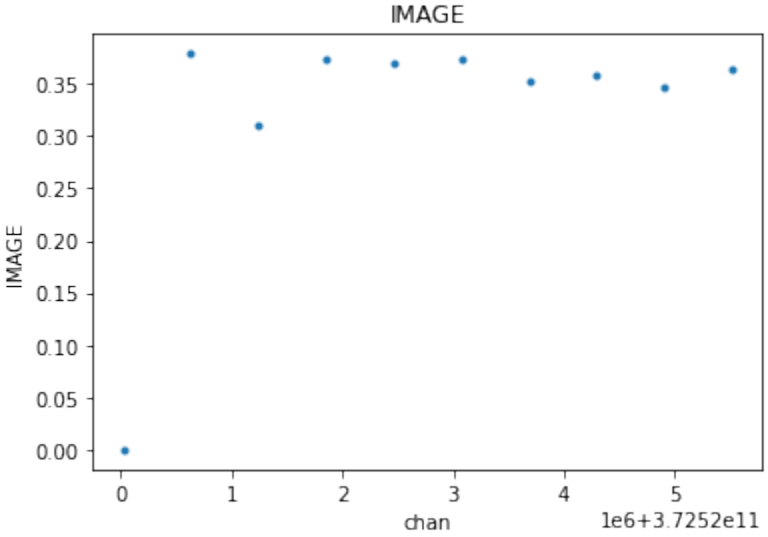
The plotting function has the same capabilities as the visibility routine, so we can also make scatter plots across coordinates and dimensions using the axis parameter. Other dimensions are collapsed by taking the max.

[9]:
```
implot(image_xds.IMAGE, axis='l')
```



Looking across channels, it is clear that channel 0 is empty. This is why the above plot across all channels is rescaled to a min value of 0.

[10]:
```
implot(image_xds.IMAGE, axis='chan')
```



5.4 Basic Manipulation

Many image operations can be easily done directly on the image xarray dataset

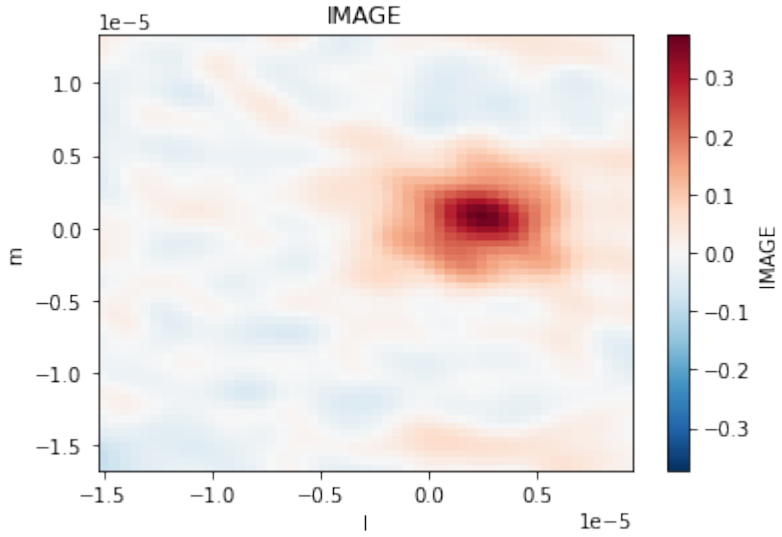
Example 1: imsubimage - copy all or part of an image to a new image

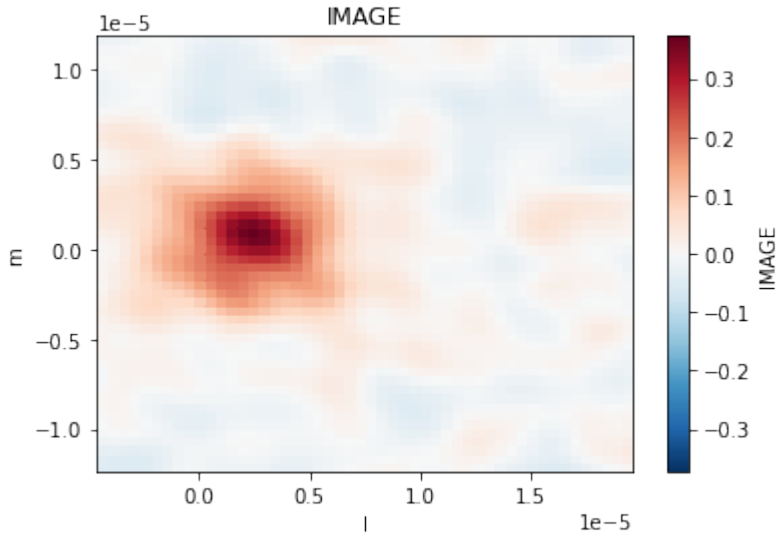
[11]:
```
# selection by coordinate values
image_xds2 = image_xds.where( (image_xds.right_ascension > 2.887905) & (image_xds.
↪right_ascension < 2.887935) &
                              (image_xds.declination > -0.60573) & (image_xds.
↪declination < -0.60570), drop=True )
```

(continues on next page)

(continued from previous page)

```
implot(image_xds2.IMAGE, chans=5)

# selection by pixel indices (50x50 pixel square)
# note that this same method works for any other dimension (stokes, channels, etc)
image_xds2 = image_xds.isel(l=range(85,135), m=range(100,150))
implot(image_xds2.IMAGE, chans=5)
```





Example 2: imtrans - reorders (transposes) the axes in the input image to the specified order

This can be done on the entire dataset, or just one variable in the dataset. Here we will do it on the whole thing which is probably best if you want to preserve your regions and masks (explained later). This will generally not affect the preview plot since it selects axes by name rather than index. That is one advantage of xarray over numpy.

[12]:
```
image_xds2 = image_xds.transpose('chan','m','time','l','pol')
print('Before\n', image_xds.data_vars)
print('\nAfter\n', image_xds2.data_vars)
```

```
Before
```

(continues on next page)

(continued from previous page)

```
 Data variables:
    AUTOMASK           (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    PB                 (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0           (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>

After
 Data variables:
    AUTOMASK           (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    IMAGE              (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    IMAGE_MASK0        (chan, m, time, l, pol) bool dask.array<chunksize=(1, 250, 1,␣
↪250, 1), meta=np.ndarray>
    IMAGE_PBCOR        (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (chan, m, time, l, pol) bool dask.array<chunksize=(1, 250, 1,␣
↪250, 1), meta=np.ndarray>
    MODEL              (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    PB                 (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    PB_MASK0           (chan, m, time, l, pol) bool dask.array<chunksize=(1, 250, 1,␣
↪250, 1), meta=np.ndarray>
    PSF                (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    RESIDUAL           (chan, m, time, l, pol) float64 dask.array<chunksize=(1, 250,␣
↪1, 250, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (chan, m, time, l, pol) bool dask.array<chunksize=(1, 250, 1,␣
↪250, 1), meta=np.ndarray>
    SUMWT              (chan, time, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

Example 3: imcollapse - collapse an image along a specified axis or set of axes of N pixels into a single pixel on each specified axis.

Again, this can be done on the entire dataset, or just one variable in the dataset. Refer to the xarray documentation for

supported aggregation functions, or write your own and call *reduce*. This will generally break the preview plot since it is expecting a 4-D cube with ra, dec, stokes, and frequency axes

[13]:
```
# one dimension collapse
image_xds2 = image_xds.sum(dim='chan')
print('One dimension collapse\n', image_xds2.data_vars)

# multi-dimension collapse
image_xds2 = image_xds.mean(dim=['l','m'])
print('\nMulti-dimension collapse\n', image_xds2.data_vars)
```

```
One dimension collapse
 Data variables:
    AUTOMASK           (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    IMAGE              (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, pol) int64 dask.array<chunksize=(250, 250, 1, 1),␣
↪meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, pol) int64 dask.array<chunksize=(250, 250, 1, 1),␣
↪meta=np.ndarray>
    MODEL              (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    PB                 (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    PB_MASK0           (l, m, time, pol) int64 dask.array<chunksize=(250, 250, 1, 1),␣
↪meta=np.ndarray>
    PSF                (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    RESIDUAL           (l, m, time, pol) float64 dask.array<chunksize=(250, 250, 1,␣
↪1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, pol) int64 dask.array<chunksize=(250, 250, 1, 1),␣
↪meta=np.ndarray>
    SUMWT              (time, pol) float64 dask.array<chunksize=(1, 1), meta=np.
↪ndarray>

Multi-dimension collapse
 Data variables:
    AUTOMASK           (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    IMAGE              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    IMAGE_MASK0        (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    IMAGE_PBCOR        (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    MODEL              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    PB                 (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    PB_MASK0           (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    PSF                (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    RESIDUAL           (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    RESIDUAL_MASK0     (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

5.5 Regions and Masks

Both regions and masks are stored as boolean arrays in the same xarray dataset alongside the rest of the image components. They always share the same dimensions as the image.

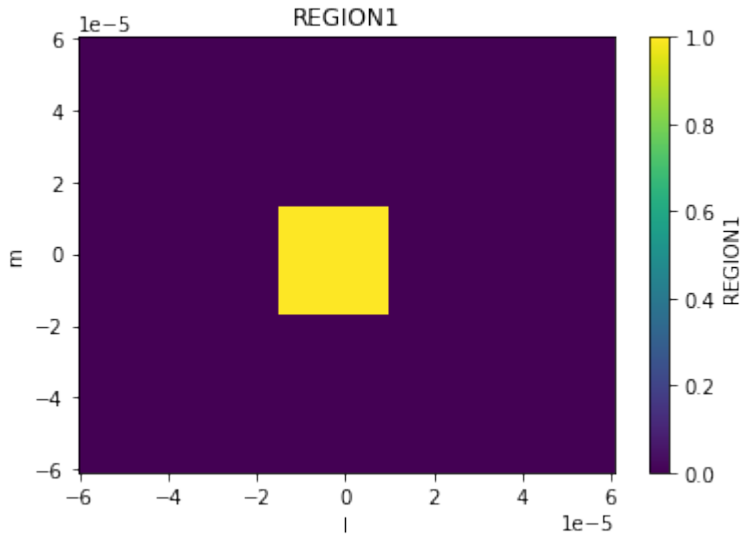
They are both now treated as the same thing, so a value of 0 means "discard this pixel" and a value of 1 means "keep this pixel" regardless of whether it is a mask or a region. In fact, any xarray data variable of boolean type can be used as a region/mask, there is nothing special about the names.

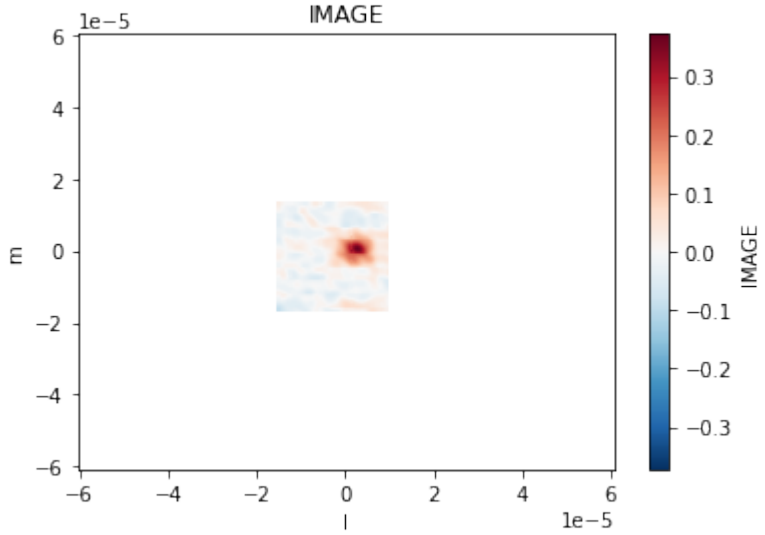
Regions/masks can be set across any dimension, so they can be per-channel and per-stokes as well as the spatial dimensions.

Regions

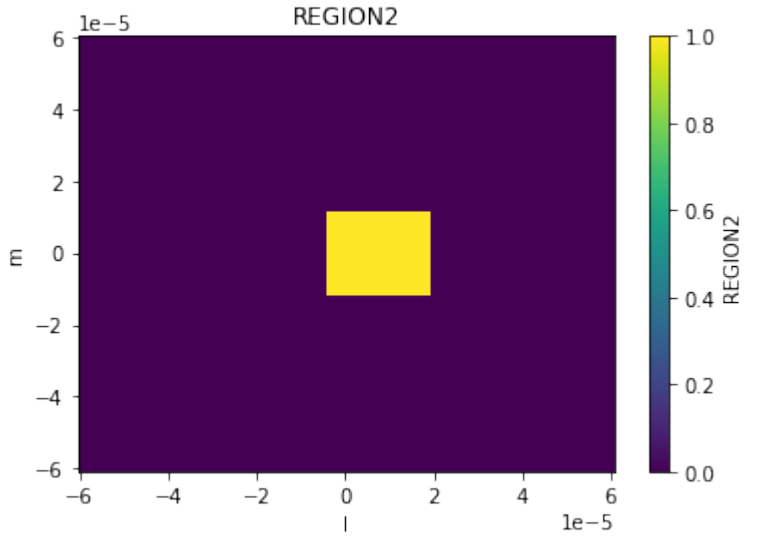
First lets create a couple examples:

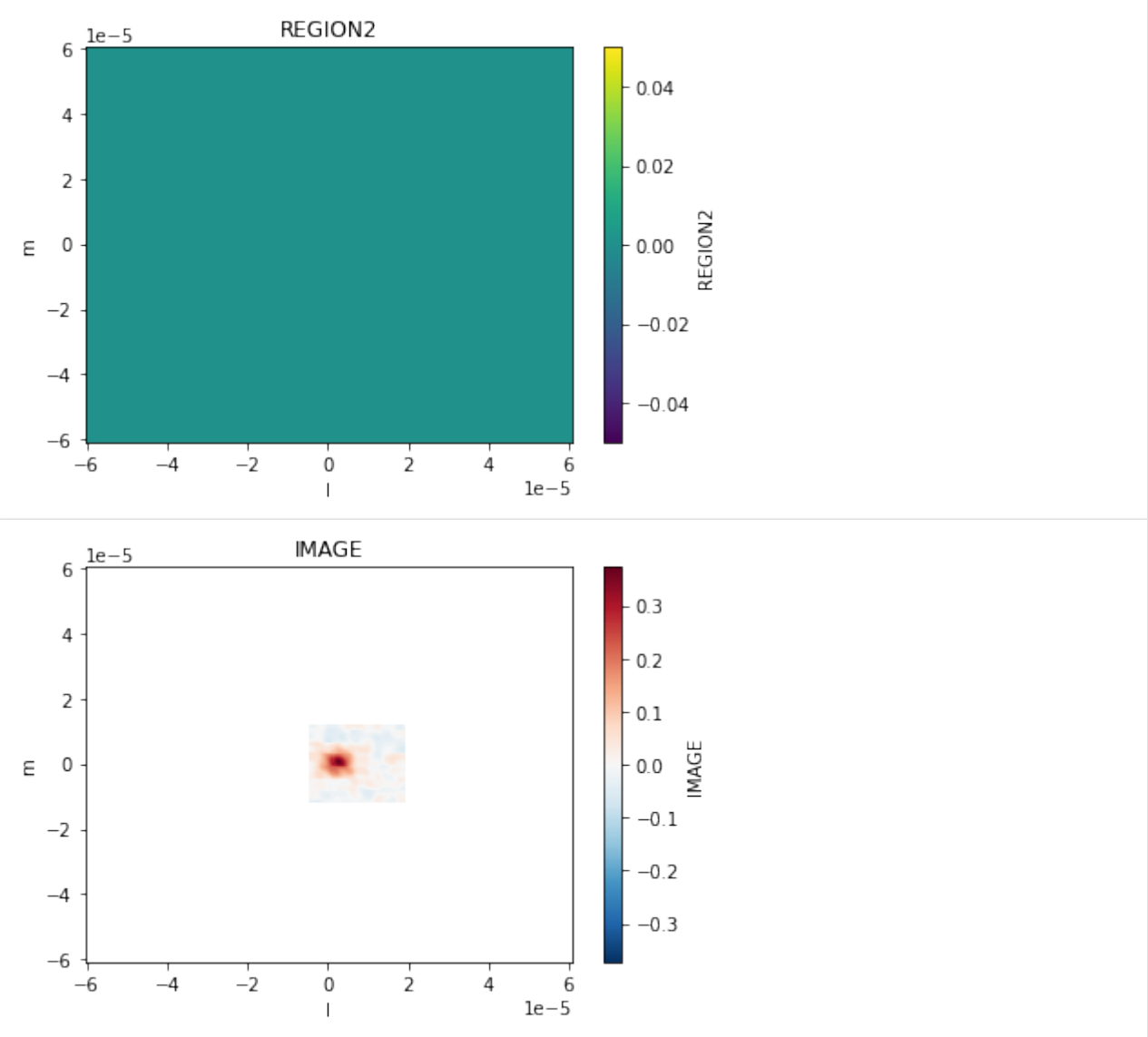
[14]:
```
from cngi.image import region

# region 1 is an ra/dec box across all channels
image_xds2 = region(image_xds, 'REGION1', ra=[2.887905, 2.887935], dec=[-0.60573, -0.
↪60570])

# lets examine what it looks like
implot(image_xds2.REGION1, chans=5)

# and combined with our image
implot(image_xds2.IMAGE.where(image_xds2.REGION1), chans=5)
```





[15]:
```
# region 2 is a pixel box across channels 3, 4 and 5
image_xds3 = region(image_xds2, 'REGION2', pixels=[[85,100],[135,150]], channels=[3,4,
↪5])

# observe the change in behavior across channels
implot(image_xds3.REGION2, chans=5)
implot(image_xds3.REGION2, chans=6)
implot(image_xds3.IMAGE.where(image_xds3.REGION2), chans=[5,6])
```





[15]:

Regions are just data variables in the xarray dataset. So we can manipulate them the same way we can any other variable.

[16]:
```
image_xds3.data_vars
```

[16]:
```
Data variables: (12/14)
    AUTOMASK           (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
```

(continues on next page)

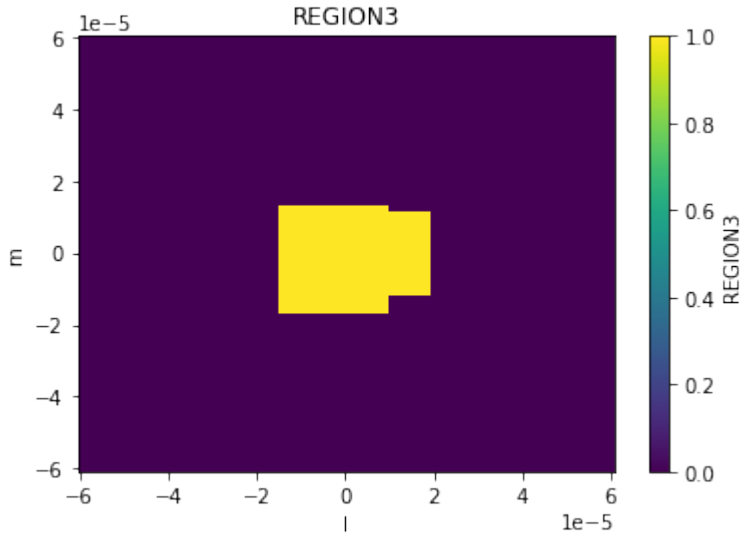
(continued from previous page)

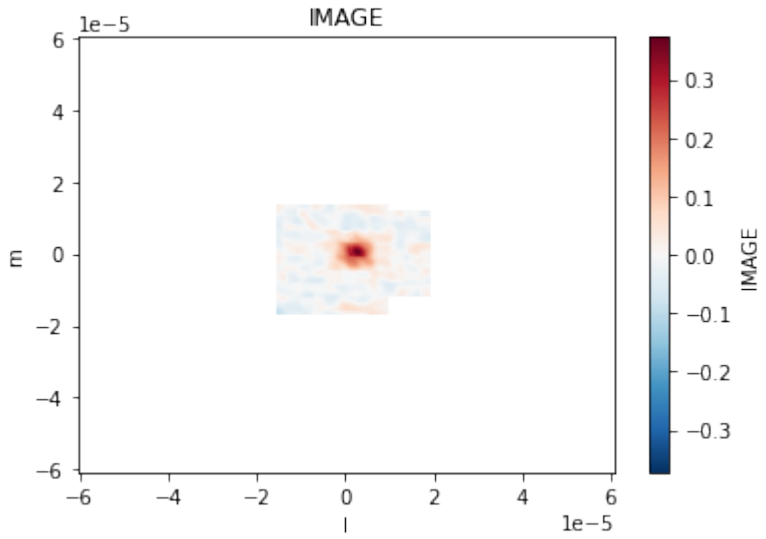
```
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    ...                 ...
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    REGION1            (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
    REGION2            (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,
↪ 1, 1), meta=np.ndarray>
```

Let's combine the first two regions in to a new one (doesn't need to be contiguous, although it is here). It is just a logical OR of two boolean arrays (obviously you could do an AND, XOR, or whatever else).

Just keep in mind that these are 4-D arrays, not 2-D.

[17]:
```
region3 = image_xds3.REGION1 | image_xds3.REGION2
image_xds4 = image_xds3.assign(dict([('REGION3', region3)]))

implot(image_xds4.REGION3, chans=5)
implot(image_xds4.IMAGE.where(image_xds4.REGION3), chans=5)
```

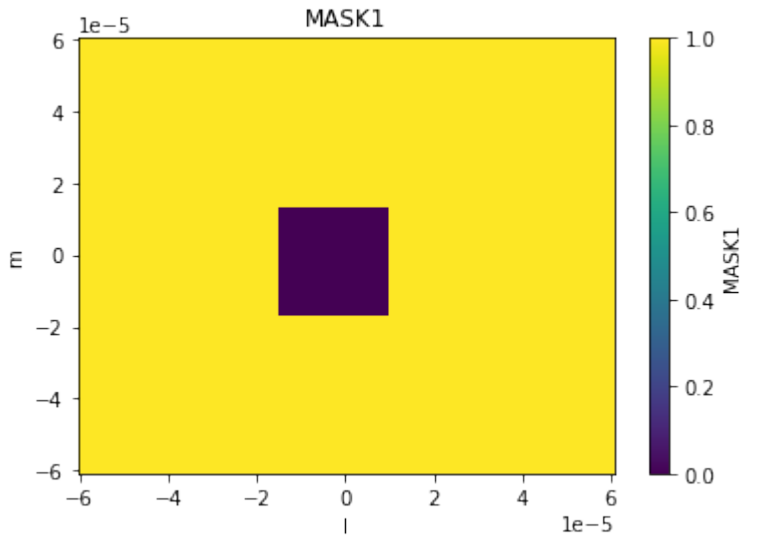


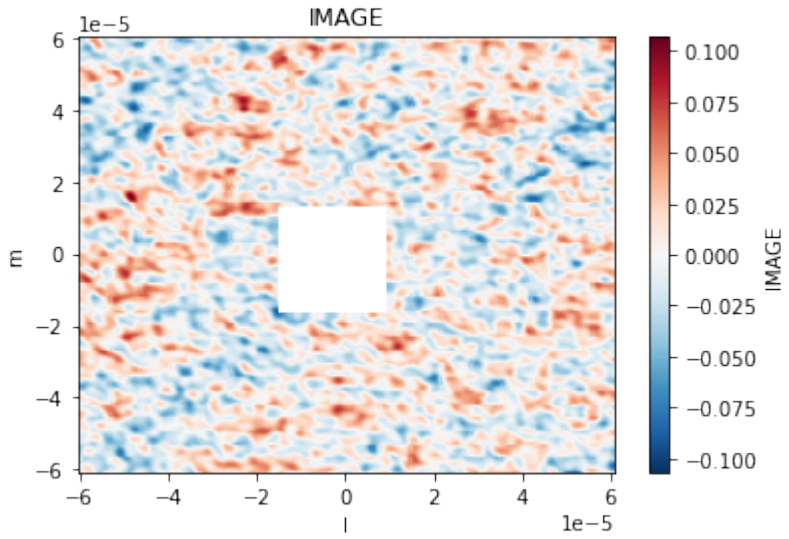


Masks

Masks are just like regions but with inverse logic used to set them. Here is the same code again using calls to mask instead of region.

[18]:
```
from cngi.image import mask

image_xds2 = mask(image_xds, 'MASK1', ra=[2.887905, 2.887935], dec=[-0.60573, -0.
↪60570])

implot(image_xds2.MASK1, chans=5)
implot(image_xds2.IMAGE.where(image_xds2.MASK1), chans=5)
```

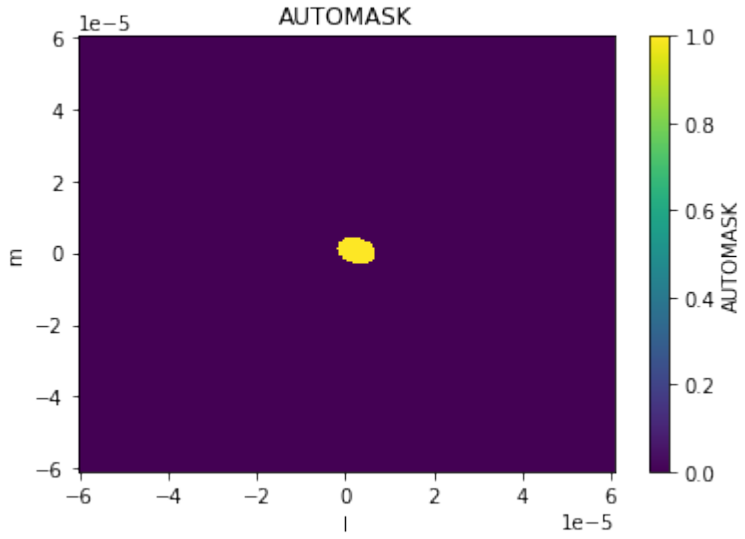


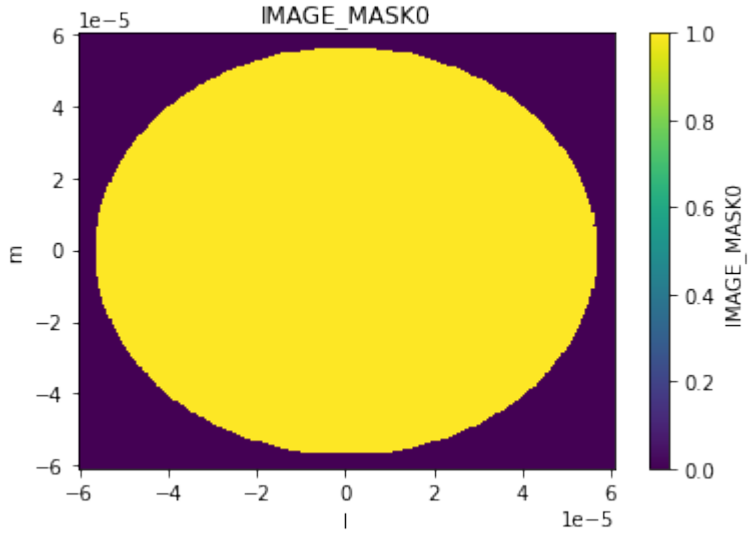


Our converted image actually had two masks in it already, the variables *mask* and *deconvolve*. Deconvolve is named differently because it is the mask set by auto-masking for the deconvolution rather than an image mask.

These actually look a lot more like regions than masks, but that's the terminology that was used. Again, they are the same thing anyway

[19]:
```
implot(image_xds2.AUTOMASK, chans=5)
implot(image_xds2.IMAGE_MASK0, chans=5)
```





Manipulation with Regions and Masks

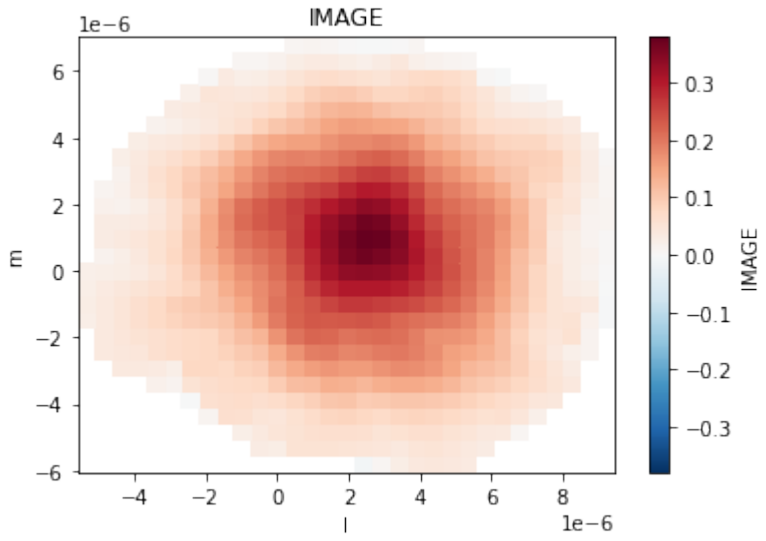
All of the previously discussed image manipulation techniques can be done with regions and masks applied. Just subselect the pixels within the region (outside the mask) first. This is done with *dataset.where(. . . , drop=True)*

Here is example 3 again with a region applied.

Example 3: imcollapse - collapse an image along a specified axis or set of axes of N pixels into a single pixel on each specified axis.

[20]:
```
# full image collapse
image_xds2 = image_xds.mean(dim=['l','m'])
print('Mean pixel values by channel : ', image_xds2.isel(pol=0).IMAGE.values, '\n')

# apply deconvolve region from auto-masking
image_xds2 = image_xds.where(image_xds.AUTOMASK, drop=True)
implot(image_xds2.IMAGE)

# collapse just the region
image_xds2 = image_xds2.mean(dim=['l','m'])
print('\nWith deconvolve mask applied : ', image_xds2.isel(pol=0).IMAGE.values)
```

```
Mean pixel values by channel :  [[0.         0.00093286 0.00062536 0.00089549 0.
↪00076526 0.00075191
  0.00072434 0.00098554 0.00082396 0.00076053]]
```




```
With deconvolve mask applied :  [[0.10387148 0.09331284 0.11133925 0.18792972 0.
↪18861415 0.11964889
  0.11996713 0.0927346  0.10202092]]
```

5.6 Beams and Smoothing

Smoothing an image involves creating and convolving beams of appropriate parameters to achieve the desired target properties in the image.

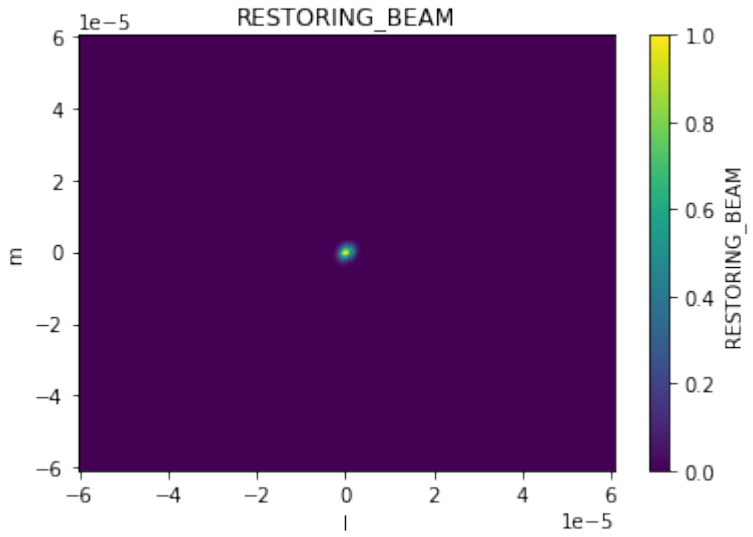
Creating Beams

Beams may be created as additional products in the image xarray Dataset. They may also be present already in parameterized form inside the attributes section.

First lets expand the parameterized per-plane beams defined in the xds attributes section. This beam is defined across the channels and polarization axes.

[21]:
```
from cngi.image import gaussian_beam

print('restoring beam parameters (chan 5) : ', image_xds.perplanebeams[5])

image_xds2 = gaussian_beam(image_xds, source='perplanebeams', name='RESTORING_BEAM')

implot(image_xds2.RESTORING_BEAM)
print('restoring beam dimensions: ', image_xds2.RESTORING_BEAM.shape)
```

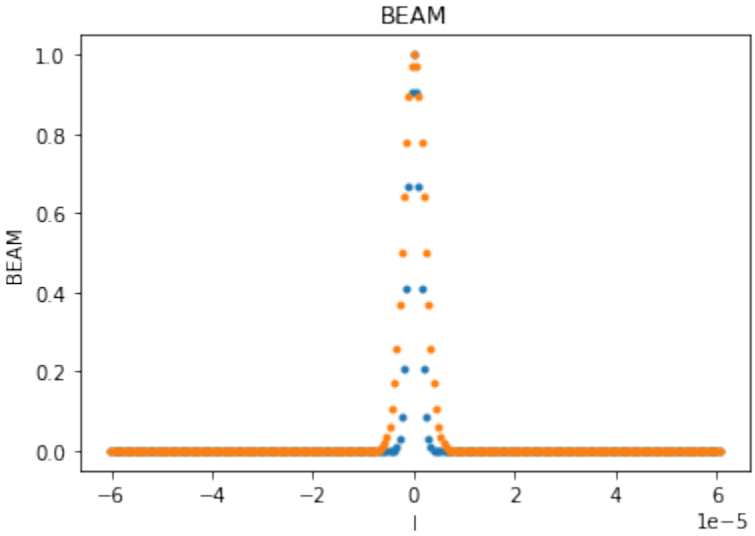
```
restoring beam parameters (chan 5) :  [[0.6526952385902405, 0.5043563842773438, -65.
↪88957977294922]]
```




```
restoring beam dimensions:  (250, 250, 10, 1)
```

Now lets create a larger beam from our own parameters. This one has only spatial dimensions.

[22]:
```
image_xds2 = gaussian_beam(image_xds2, source=[1., 1., 30], name='BEAM')

implot(image_xds2.RESTORING_BEAM, 'l', drawplot=False)
implot(image_xds2.BEAM, 'l', overplot=True)
print('custom beam dimensions: ', image_xds2.BEAM.shape)
```




```
custom beam dimensions:  (250, 250)
```

Smoothing with Beams

The smooth function can take an existing beam and convolve it across the image to produce a new output image.

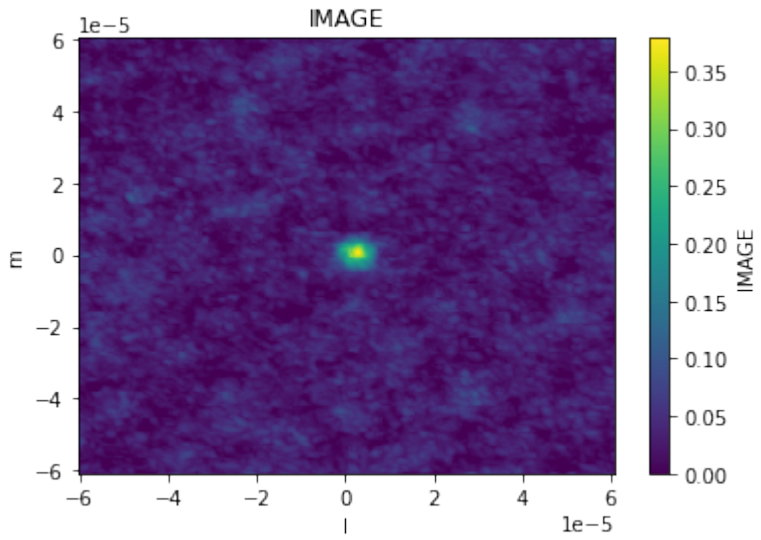
[23]:
```
from cngi.image import smooth

# smooth with the restoring beam
```

(continues on next page)

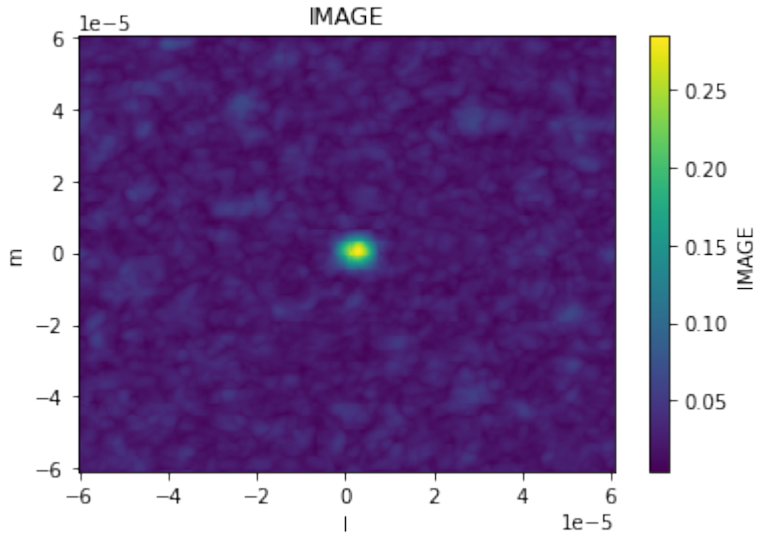
(continued from previous page)

```
image_xds3 = smooth(image_xds2, kernel='RESTORING_BEAM')

print('original image')
implot(image_xds2.IMAGE)

print('smooth image')
implot(image_xds3.IMAGE)
```

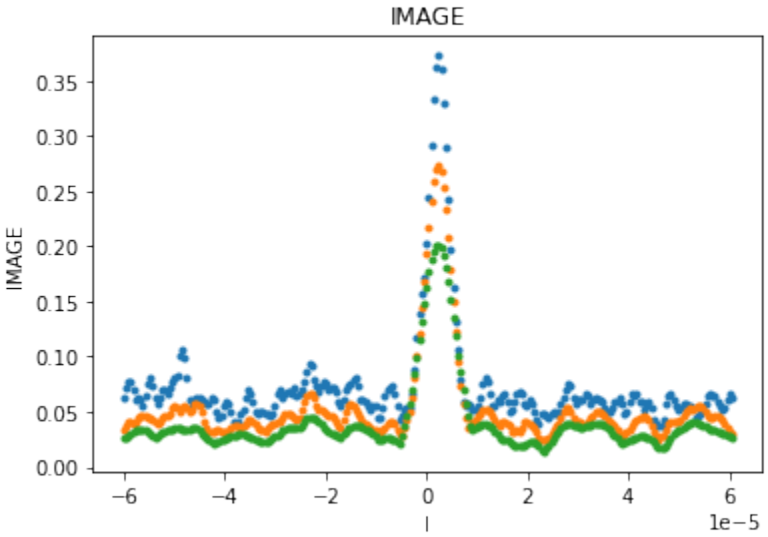
```
original image
```




```
smooth image
```

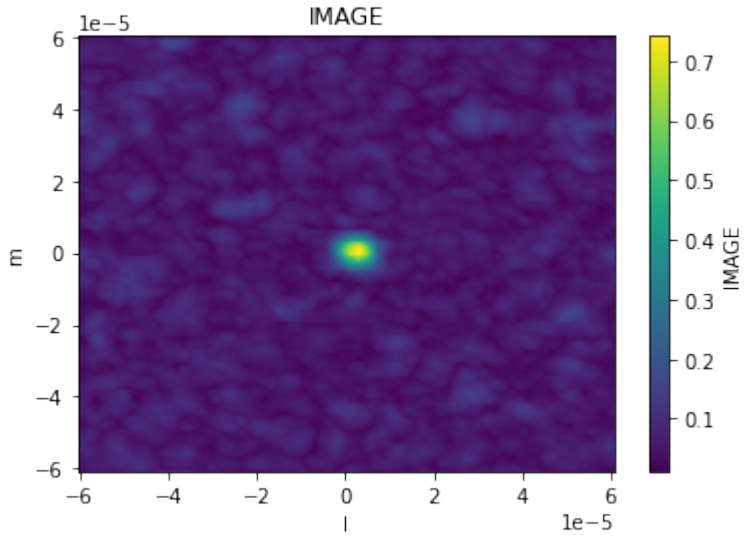


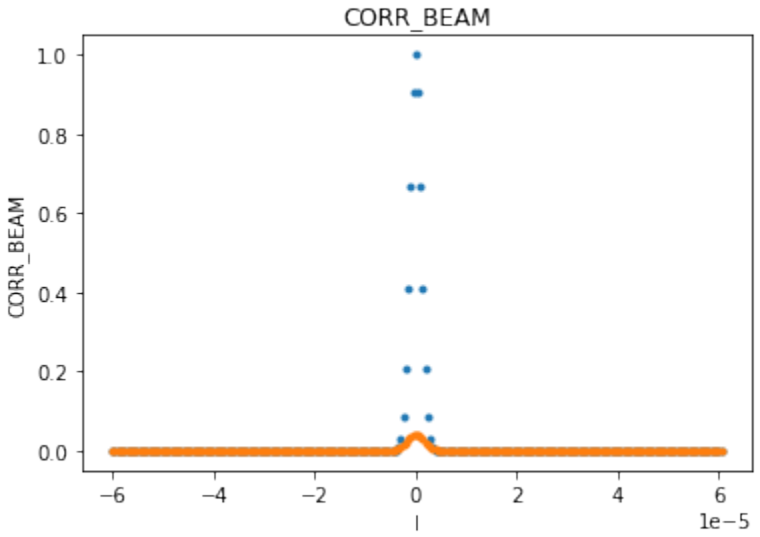
[24]:
```
# smooth with our larger custom beam
image_xds4 = smooth(image_xds2, kernel='BEAM')

implot(image_xds.IMAGE, axis='l', chans=5, drawplot=False)
implot(image_xds3.IMAGE, axis='l', chans=5, overplot=True, drawplot=False)
implot(image_xds4.IMAGE, axis='l', chans=5, overplot=True)
```



If the image has already been smoothed by a previous beam, the smooth function can take the parameters of a new desired beam and construct the appropriate correcting beam to produce that target. The correcting beam used for smoothing is saved in the output xarray Dataset.

[25]:
```
image_xds3 = smooth(image_xds2, kernel='gaussian', size=(1., 1., 60.), current=(0.
↪65305, 0.50467, -66.), name='CORR_BEAM')

implot(image_xds3.IMAGE)
implot(image_xds3.RESTORING_BEAM, axis='l', chans=5, drawplot=False)
implot(image_xds3.CORR_BEAM, axis='l', overplot=True)
```





5.7 Moments

The `moments` function is used to compute moments on the specified image xds data_var. Moments are stored as additional data_vars in the returned xds.

[26]:
```
from cngi.image import moments, implot

image_xds2 = moments(image_xds, moment=[-1,0,1,2,3,4,5,6,7,8,9,10,11], axis='chan')

print(image_xds2.data_vars)
```

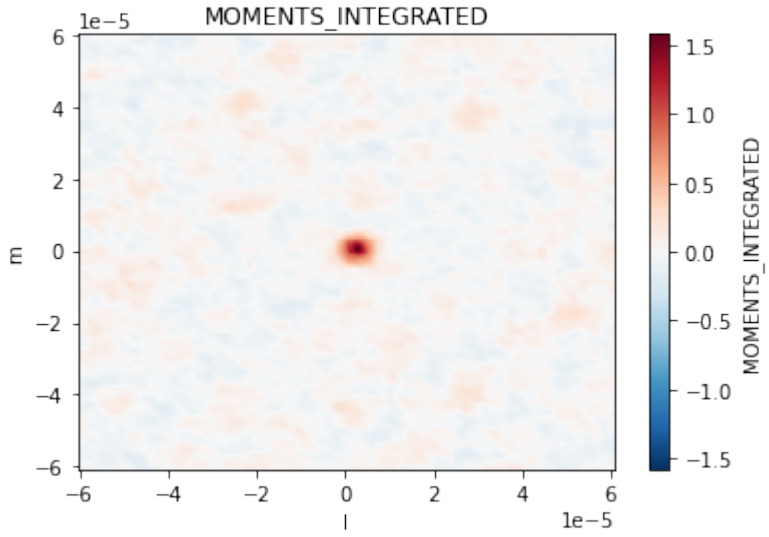
```
Data variables: (12/25)
    AUTOMASK                          (l, m, time, chan, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1, 1), meta=np.ndarray>
    IMAGE                             (l, m, time, chan, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0                       (l, m, time, chan, pol) bool dask.array
↪<chunksize=(250, 250, 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR                       (l, m, time, chan, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0                 (l, m, time, chan, pol) bool dask.array
↪<chunksize=(250, 250, 1, 1, 1), meta=np.ndarray>
    MODEL                             (l, m, time, chan, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1, 1), meta=np.ndarray>
    ...                                ...
    MOMENTS_RMS                       (l, m, time, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1), meta=np.ndarray>
    MOMENTS_ABS_MEAN_DEV              (l, m, time, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1), meta=np.ndarray>
    MOMENTS_MAXIMUM                   (l, m, time, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1), meta=np.ndarray>
    MOMENTS_MAXIMUM_COORD             (l, m, time, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1), meta=np.ndarray>
    MOMENTS_MINIMUM                   (l, m, time, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1), meta=np.ndarray>
```

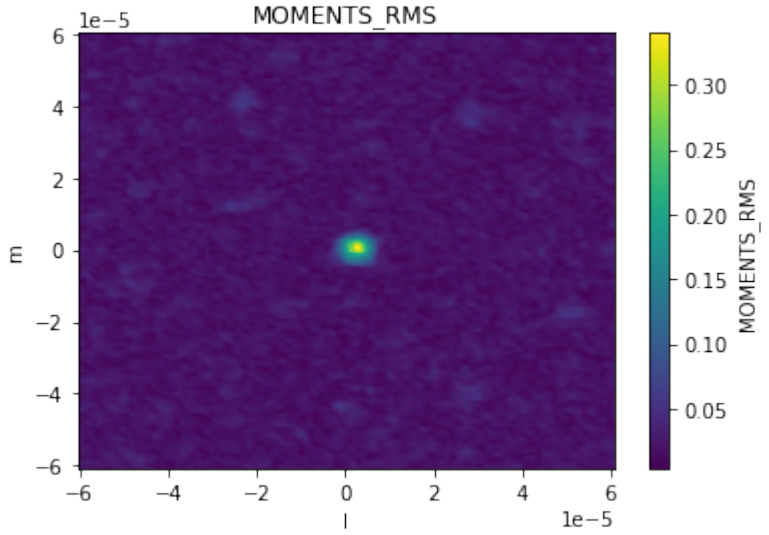
(continues on next page)

(continued from previous page)

```
    MOMENTS_MINIMUM_COORD              (l, m, time, pol) float64 dask.array
↪<chunksize=(250, 250, 1, 1), meta=np.ndarray>
```

[27]:
```
implot(image_xds2.MOMENTS_INTEGRATED)
implot(image_xds2.MOMENTS_RMS)
```





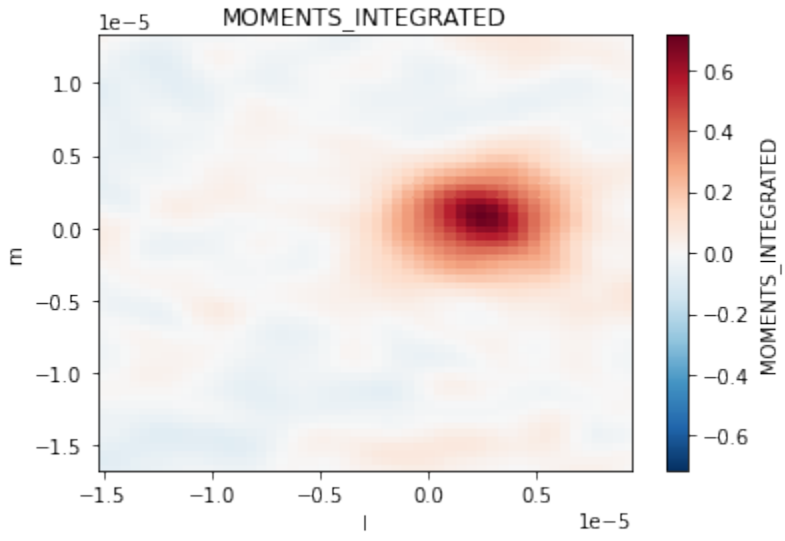
Use basic manipulation, regions, or masks before hand to select a subset of the image for moments

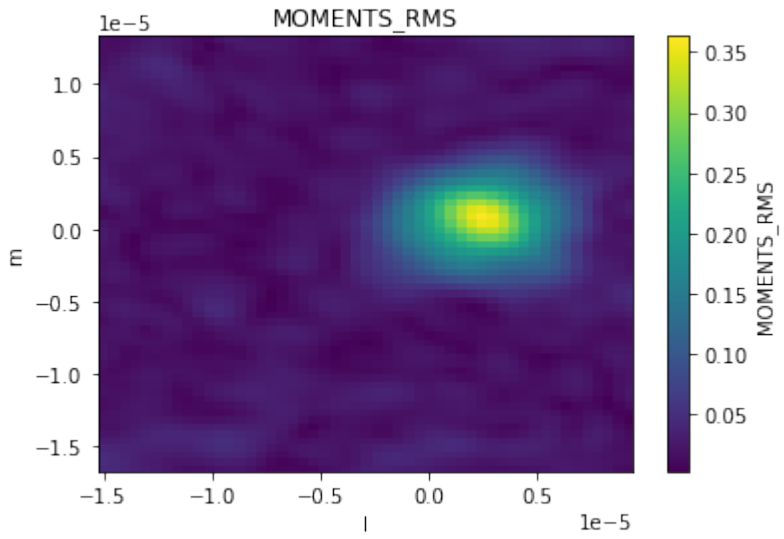
[28]:
```
# Example for creating selected moment maps for selected channels
from cngi.image import region

# region 1 is an ra/dec box across all channels
image_xds2 = region(image_xds, 'REGION1', ra=[2.887905, 2.887935], dec=[-0.60573, -0.
↪60570]).isel(chan=[4,5,6,7])
image_xds2 = moments(image_xds2.where(image_xds2.REGION1, drop=True), moment = [0,6])

implot(image_xds2.MOMENTS_INTEGRATED)
```

(continues on next page)

(continued from previous page)

```
implot(image_xds2.MOMENTS_RMS)
```





5.8 Statistics

The `statistics` function is used to compute statistics on the specific image xds data_var. The results are placed in the attributes section of the returned xds.

By default this is done in a lazy fashion so as not to break the xds DAG. To see the actual value of a statistic, the `.values` method is needed.

[29]:
```
from cngi.image import statistics

image_xds2 = statistics(image_xds, dv='IMAGE', name='statistics')

print('computed statistics:\n -', '\n - '.join(list(image_xds2.statistics.keys())))
```

(continues on next page)

(continued from previous page)

```
print('\nlazy statistic (mean) :', image_xds2.statistics['mean'])
print('\ncomputed statistic (mean) :', image_xds2.statistics['mean'].values.item())
```

```
computed statistics:
 - max
 - maxpos
 - maxposf
 - mean
 - medabsdevmed
 - median
 - min
 - minpos
 - minposf
 - npts
 - q1
 - q3
 - quartile
 - rms
 - sigma
 - sum
 - sumsq
 - blc
 - blcf
 - trc
 - trcf

lazy statistic (mean) : <xarray.DataArray 'IMAGE' ()>
dask.array<mean_agg-aggregate, shape=(), dtype=float64, chunksize=(), chunktype=numpy.
↪ndarray>

computed statistic (mean) : 0.0007265257783630225
```

Optionally, `compute=True` can be passed to the `statistics` function. This will break the DAG and could be costly in terms of performance.

[30]:
```
image_xds2 = statistics(image_xds, dv='IMAGE', name='statistics', compute=True)

for kk in image_xds2.statistics.keys():
   print(kk + ': ', image_xds2.statistics[kk])
```

```
max:  0.3788154721260071
maxpos:  [120, 127, 0, 1, 0]
maxposf:  ['11:01:51.83654673', '-34.42.17.16599958', '2012-11-19T07:56:26.544000626',
↪ 372520632933.7965, 1.0]
mean:  0.0007265257783630225
medabsdevmed:  0.013578892219811678
median:  0.0
min:  -0.11158199608325958
minpos:  [208, 245, 0, 2, 0]
minposf:  ['11:01:51.12295091', '-34.42.05.36588434', '2012-11-19T07:56:26.544000626',
↪ 372521243263.95557, 1.0]
npts:  625000
q1:  -0.013601248385384679
q3:  0.013555713929235935
quartile:  0.027156962314620614
rms:  0.024755708055986946
```

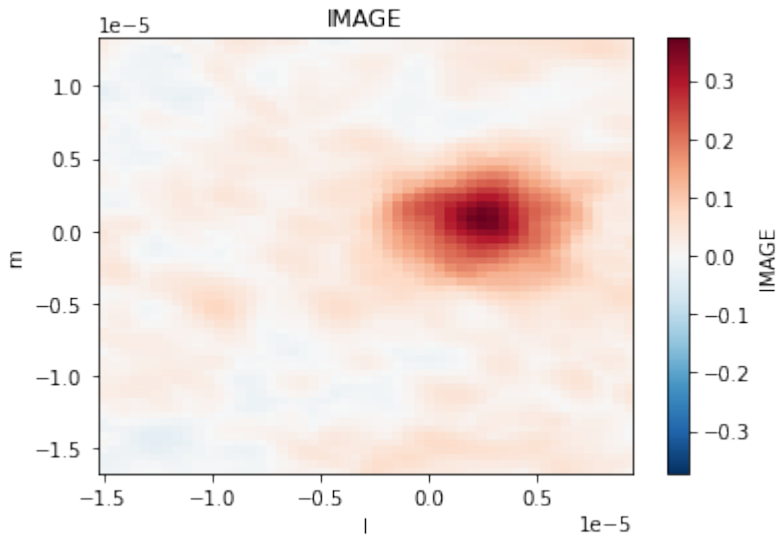
(continues on next page)

(continued from previous page)

```
sigma:  0.024745044789747924
sum:  454.078611476889
sumsq:  383.02817584578554
blc:  [0 0 0 0 0]
blcf:  ['11:01:52.80971154', '-34.42.29.86573768', '2012-11-19T07:56:26.544000626',␣
↪372520022603.63745, 1.0]
trc:  [249 249   0   9   0]
trcf:  ['11:01:50.79048222', '-34.42.04.96574187', '2012-11-19T07:56:26.544000626',␣
↪372525515575.069, 1.0]
```

Use basic manipulation, regions, or masks before hand to select a subset of the image for statistics

[31]:
```
# Example of select region and print the statistics.
from cngi.image import region

# region 1 is an ra/dec box across all channels
image_xds2 = region(image_xds, 'REGION1', ra=[2.887905, 2.887935], dec=[-0.60573, -0.
↪60570]).isel(chan=[4,5,6,7])
image_xds2 = statistics(image_xds2.where(image_xds2.REGION1, drop=True), dv='IMAGE',␣
↪name='statistics', compute=True)

implot(image_xds2.IMAGE)

for kk in image_xds2.statistics.keys():
   print(kk + ': ', image_xds2.statistics[kk])
```




```
max:  0.3726564943790436
maxpos:  [14, 36, 0, 1, 0]
maxposf:  ['11:01:51.83654673', '-34.42.17.16599958', '2012-11-19T07:56:26.544000626',
↪ 372523074254.43274, 1.0]
mean:  0.016538425474854918
medabsdevmed:  0.019604451023042202
median:  0.0023901931708678603
min:  -0.0871538370847702
minpos:  [50, 1, 0, 1, 0]
minposf:  ['11:01:51.5446073', '-34.42.20.66598387', '2012-11-19T07:56:26.544000626',␣
↪372523074254.43274, 1.0]
```

(continues on next page)

(continued from previous page)

```
npts:  12648
q1:  -0.01503333286382258
q3:  0.025363899767398834
quartile:  0.040397232631221414
rms:  0.060676759432310934
sigma:  0.05837935952046069
sum:  209.178005405965
sumsq:  46.565751222092246
blc:  [0 0 0 0 0]
blcf:  ['11:01:51.95007945', '-34.42.20.76599394', '2012-11-19T07:56:26.544000626',␣
↪372522463924.2737, 1.0]
trc:  [50 61  0  3  0]
trcf:  ['11:01:51.54461237', '-34.42.14.66598387', '2012-11-19T07:56:26.544000626',␣
↪372524294914.75085, 1.0]
```

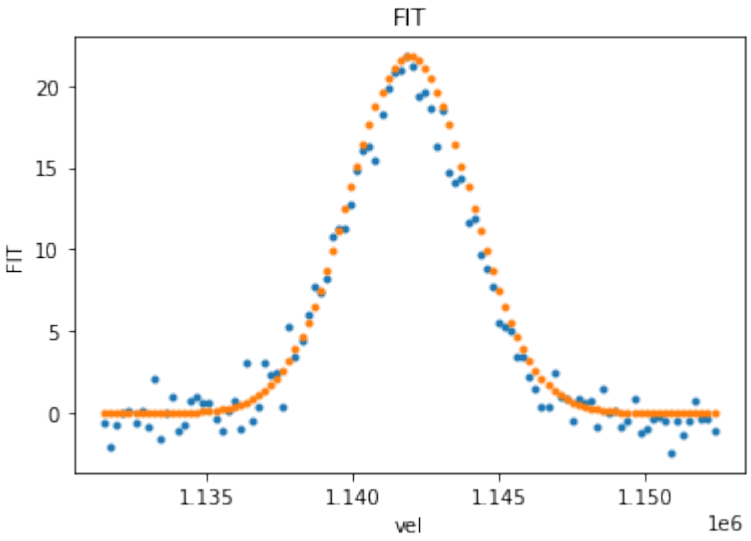
5.9 Spectral Line Fitting

Adapted from https://github.com/emilyripka/BlogRepo/blob/master/181119_PeakFitting.ipynb Dave Mehringer 2021mar01

[32]:
```
# Create a blank image in CASA 6
import casatools
import numpy as np

myia = casatools.image()
nchan = 100
myia.fromshape("gaussfit.im",[20,20,nchan,1], overwrite=True)

# insert the model spectrum
x_vals = np.linspace(0,nchan-1,nchan)
y_vals = 20*(np.exp((-1.0/2.0)*(((x_vals-(nchan/2))/10)**2)))
pix = myia.getchunk()
pix[10, 10, :, 0] = y_vals
myia.putchunk(pix)

# add some noise
myia.addnoise()
myia.done()

# convert to CNGI image
image_xds10 = convert_image('gaussfit.im')
```

```
converting Image...
processed image in 0.32384706 seconds
```

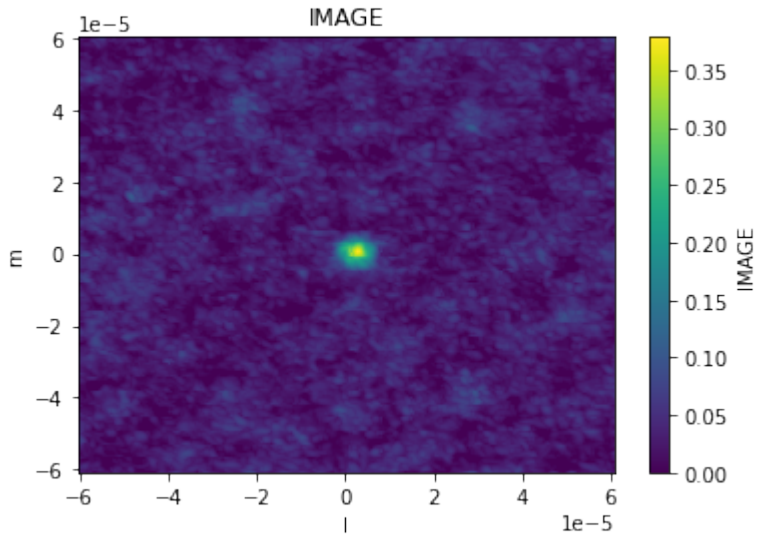
[33]:
```
# convert freqs to velocities
from cngi.image import spec_fit, implot

nxds = spec_fit(image_xds10, dv='IM', pixel=(10,10), name='FIT')

implot(nxds.IM[10,10,0,:,0], axis='vel', drawplot=False)
implot(nxds.FIT, axis='vel', overplot=True)
```

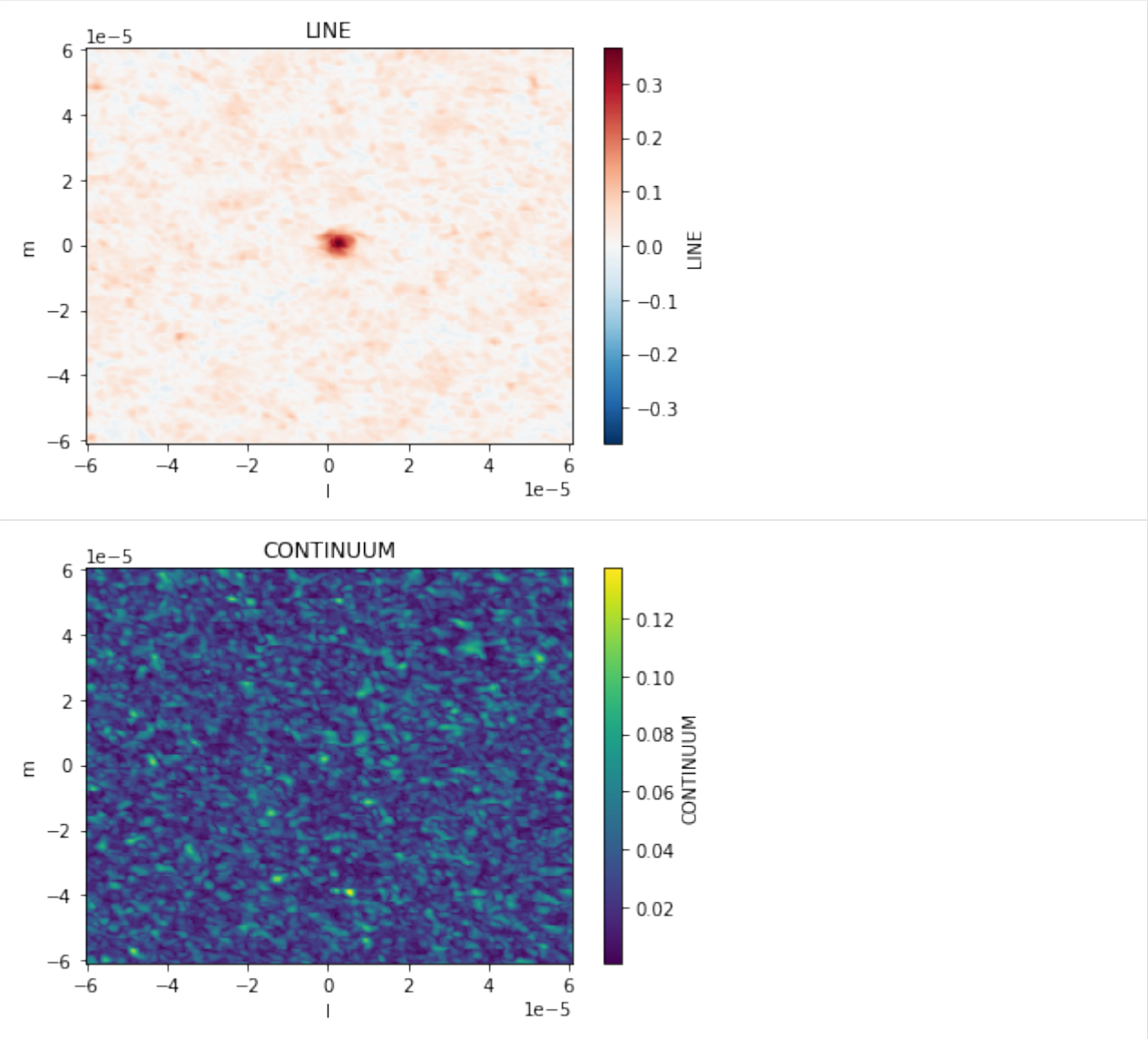


5.10 Continuum Subtraction

The image cont_sub function estimates and subtracts continuum emission from an image cube. The line emission and continuum images are stored as additional data_vars in the returned xds

[34]:
```
from cngi.image import cont_sub

# Fit a second order polynomial (fitorder=2) to channels 1-3 and 7-9 to an RA x Dec x␣
↪Frequency x Stokes cube.
image_xds2 = cont_sub(image_xds, dv='IMAGE', fitorder=2, chans=[1,2,3,7,8,9],␣
↪linename='LINE', contname='CONTINUUM')

implot(image_xds2.IMAGE)
implot(image_xds2.LINE)
implot(image_xds2.CONTINUUM)
```





The accuracy of the polynomial fit is stored in the attributes section of the returned xds. By default this is not computed (left as lazy DAG). Use the `values` method to see the results. Alternatively, `compute=True` may be passed to the `spec_fit` function.

[35]:
```
print(image_xds2.line)
```

```
{'rms_error': <xarray.DataArray ()>
dask.array<pow, shape=(), dtype=float64, chunksize=(), chunktype=numpy.ndarray>, 'min_
↪max_error': [<xarray.DataArray ()>
dask.array<nanmin-aggregate, shape=(), dtype=float64, chunksize=(), chunktype=numpy.
↪ndarray>, <xarray.DataArray ()>
dask.array<nanmax-aggregate, shape=(), dtype=float64, chunksize=(), chunktype=numpy.
↪ndarray>], 'bw_frac': 0.6, 'freq_frac': <xarray.DataArray 'chan' ()>
array(0.88888889)}
```

[36]:
```
image_xds2 = cont_sub(image_xds, dv='IMAGE', fitorder=2, chans=[1,2,3,7,8,9],␣
↪linename='LINE', contname='CONTINUUM', compute=True)
```

(continues on next page)

(continued from previous page)

```
print(image_xds2.line)
```

```
{'rms_error': 0.0132873388624595, 'min_max_error': [9.583802056817303e-08, 0.
↪06687562817485812], 'bw_frac': 0.6, 'freq_frac': 0.8888888888888888}
```

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/calibration.ipynb

CHAPTER

SIX

CALIBRATION

The initial demonstration of calibration in CNGI/ngCASA is centered around a streamlined implementation of on-the-fly self-calibration for synthesis visibility data. The purpose of the prototype implementation of self_cal is to demonstrate the use and throughput of the dask-based parallelization framework in the calibration context, and begin to explore how some essential components of general synthesis calibration (gain solving, application) might appear in python, and built upon the xarray visibility data structures. As such, this prototype is limited in several ways compared to the more conventional approach implemented in traditional CASA. In fact, there is no direct single-execution analog of ngCASA self_cal available within CASA, where self-calibration involves a sequence of executions of gaincal and applycal (and includes multiple passes through the data). In contrast, ngCASA self_cal is a single-pass implementation that takes an input data group (visibilities, weights, flags, and model), performs an antenna-based gain solution (at the native time granularity of the data), applies the result to the input data, and returns a calibrated output data group. Most notably, there is not yet a way to examine the solved-for gain solutions themselves. This is mainly a practical consequence of not yet having designed the calibration solution container (caltable), nor the mechanisms for filling and examining it. Rather, the data product (for now), are the corrected visibilities, appropriate for comparison with the input visibilities and for imaging. More comprehensive tools for generating, managing, and examining calibration will be designed and implemented in ngCASA in future. Nonetheless, it is thought that a single-pass implementation of the sort demonstrated here will have a role in the processing of data from the larger arrays of the future (ngVLA, etc.) since self-calibration is likely the most I/O- and computationally-intensive calibration use case, insofar as it must process the (usually) most-voluminous science target visibilities in any synthesis visibility dataset. It will also be possible to integrate this self-calibration mechanism within the imaging regime, e.g., to implement a form a difference mapping. Thus, this demonstration is well-tuned to the specific question of applicability of the CNGI/ngCASA framework to the calibration domain.

Note that the self_cal function has not undergone the degree of performance optimization that imaging/gridding has, and is therefore not intended as a demonstration of framework performance.

The ngCASA self_cal function ingests an appropriately time-chunked xarray dataset including (possibly nominally calibrated) visibility data, weights, flags, and model. For each dask chunk in the xarray data group, the input visibility data, model, and weights are conditioned for the solve by (a) zeroing the weights for flagged or absent data, (b) slicing out just the parallel-hands, (c) forming the ratio of visibility and model (including weight update), (d) weighted averaging over frequency channels, (e) (optionally) combining the parallel-hand correlations for a single-pol ('T') solution (including weight update), (f) (optionally) weighted averaging of the time axis up to the virtual dask chunking (including weight update), (g) (optionally) dividing by the visibility amplitudes for phase-only solutions (including weight update). This results in data and weight arrays properly prepared and maximally collapsed for sliced ingestion within the solve loop. Then, for each timestamp and polarization, the solve loop will: (a) detect the available antenna-based constraints, zeroing the weights for all baselines involving antennas with insufficient data according to minblperant, (b) calculate a first guess for the gains based on the available baselines to the specified reference antenna, (c) perform the scalar gain solve via scipy.optimize.least_squares, supplying a weighted-residual calculation function that embodies the (scalar, for now) multiplicative algebra of the visibilities and gains, (d) derive solution error information, and (e) store the solved-for gains in a (temporary) array. Upon completion of the solve loop: (a) phase-only gains are enforced to have unit amplitude (if necessary), (b) the user-specified SNR threshold is applied, and (c) the original data arrays (all channels, correlations, times) are corrected. These results for all dask chunks are then aggregated into a new xarray data group.

Current notable limitations:

- Only per-integration or per-chunk (dask) solution intervals
- No spw combine for solve (requires higher-level management of xarray datasets).
- No refantmode support (mostly inconsequential in this context)
- No amplitude-only solution
- No robust solving (e.g., L1) support (scipy.optimize.least_squares options in this area not yet exposed)
- No solution normalization (e.g. to preserve input f.d. scale)
- No global refant reconcilation (at worst, cross-hand phase uniformity may be compromised for 'G' solving; otherwise this is inconsequential for OTF self-calibration).
- No solution normalization (e.g. to preserve input f.d. scale)
- No gain interpolation in the calibration apply (this is inconsequential for per-integration solutions)o No non-trivial solution mapping on apply (e.g., spwmap, etc.)
- No access to the raw gain solutions (requires design and implementation of a caltable)
- No prior calibration apply support

6.1 Installation

In this section, we install the prototype software and download the dataset.

[1]:
```
import os

os.system("pip install cngi-prototype==0.0.91")

!gdown -q --id 1esqPjOVVr-fuox78lqL7kxXv3Dt-T7Ox
!unzip selfcal_sim1_1scan.vis.zarr.zip > /dev/null

%matplotlib widget
import xarray as xr
xr.set_options(display_style="html")
print('complete')
```

```
complete
```

6.2 Run self_cal

In this example, we are self-calibrating simulated full-polarization point-source visibility data. The putative observation (based on a real ALMA dataset) had 47 antennas in one spectral window, with four correlations in the circular basis. The data are corrupted by a systematic set of scalar multiplicative gain errors that are constant in time, frequency, and polarization. Gaussian noise at a typical scale (appropriate for the recorded bandwidth and integration time, and an SEFD of ~8400 Jy). This example is perhaps atypical for self-cal in the sense that the data are not nominally calibrated to any degree, but since we can asumme a point source model, we can expect a good result. In this case, we are assuming a 3 Jy (total intensity) point source model, which was stored with the imported data. Since the gain errors affect all correlations in a systematic way, we can expect the calibration derived from the parallel hands to also properly calibrate the cross-hands, and thereby reveal the linear polarization of the point source (which we "suspect" to be 13%...).

[2]:
```
import xarray as xr
from cngi.dio import read_vis, read_image, write_vis
from ngcasa.calibration import self_cal
from cngi.vis import chan_average, time_average, apply_flags
import dask
xr.set_options(display_style="html")


# Import the data, non-trivially chunking it (virtually) in time (only)

#Dims  (time: 300,baseline: 1128, chan: 11, pol: 4)
#Zarr chunks (time: 10, baseline: 1128, chan: 2, pol: 4)
vis_mxds = read_vis("selfcal_sim1_1scan.vis.zarr",chunks={'time':150,'chan':12})

# apply flags (this isn't strictly necessary, since self-cal will handle flags
#    internally)
vis_mxds_flagged = apply_flags(vis_mxds,vis='xds0')

# Select the 0th data group in xds0 for processing
#   (the corrected data will be added to this data group)
sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0
sel_parms['data_group_out_id'] = 0

# Arrange solving parameters
#   gaintype='T' (single pol solution for all correlations)
#   solint='int' (per-integration solutions)
#   refant_id=4  (pick a refant, any refant)
#   (let other parameters default)
solve_parms= {}
solve_parms['gaintype']='T'
solve_parms['solint']='int'
solve_parms['refant_id']=4
# solve_parms['phase_only']=False
# solve_parms['minsnr']=0.0
# solve_parms['minblperant']=4

# Run the self_cal function
sc_vis_mxds = self_cal(vis_mxds_flagged, solve_parms, sel_parms)

# Export the corrected data to a new zarr
write_vis(sc_vis_mxds,"cor_selfcal_sim1_1scan.vis.zarr")
#sc_vis_mxds.xds0
```

```
overwrite_encoded_chunks True
######################### Start self_cal ##########################
Setting default phase_only  to  False
Setting default minsnr  to  0.0
Setting default minblperant  to  4
Setting default ginfo  to  False
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out [' data ']  to  DATA
Setting default data_group_out [' flag ']  to  FLAG
Setting default data_group_out [' uvw ']  to  UVW
Setting default data_group_out [' weight ']  to  DATA_WEIGHT
Setting default data_group_out [' corrected_data ']  to  CORRECTED_DATA
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_out [' corrected_data_weight ']  to  CORRECTED_DATA_WEIGHT
Setting default data_group_out [' corrected_flag ']  to  CORRECTED_FLAG
Setting default data_group_out [' flag_info ']  to  FLAG_INFO
######################### Created self_cal graph #########################
Time to store and execute graph for  xds0 write_zarr 93.75184869766235
Time to store and execute graph for  ANTENNA write_zarr 0.026245832443237305
Time to store and execute graph for  ASDM_ANTENNA write_zarr 0.03313755989074707
Time to store and execute graph for  ASDM_CALATMOSPHERE write_zarr 0.2484433650970459
Time to store and execute graph for  ASDM_CALWVR write_zarr 0.09831571578979492
Time to store and execute graph for  ASDM_CORRELATORMODE write_zarr 0.
↪03370928764343262
Time to store and execute graph for  ASDM_RECEIVER write_zarr 0.027709245681762695
Time to store and execute graph for  ASDM_SBSUMMARY write_zarr 0.08381080627441406
Time to store and execute graph for  ASDM_SOURCE write_zarr 0.06781673431396484
Time to store and execute graph for  ASDM_STATION write_zarr 0.018327951431274414
Time to store and execute graph for  FEED write_zarr 0.04464125633239746
Time to store and execute graph for  FIELD write_zarr 0.05102205276489258
Time to store and execute graph for  FLAG_CMD write_zarr 0.08463907241821289
Time to store and execute graph for  OBSERVATION write_zarr 0.028470993041992188
Time to store and execute graph for  POLARIZATION write_zarr 0.011534690856933594
Time to store and execute graph for  PROCESSOR write_zarr 0.02015399932861328
Time to store and execute graph for  SOURCE write_zarr 0.05385231971740723
Time to store and execute graph for  SPECTRAL_WINDOW write_zarr 0.05361151695251465
Time to store and execute graph for  STATE write_zarr 0.030668258666992188
Time to store and execute graph for  WEATHER write_zarr 0.07512664794921875
```

6.3 Plot results I

In this section, we plot a single channel in a selected time range. This shows the raw SNR of the data.

[3]:
```
import xarray as xr
import matplotlib.pylab as plt
import numpy as np
from cngi.vis import chan_average
from cngi.dio import read_vis
import dask

# Import the corrected data
sc_vis_mxds = read_vis("cor_selfcal_sim1_1scan.vis.zarr",chunks={'time':150})

# Select a single channel and a subset of time for plotting
xds = sc_vis_mxds.xds0.isel(chan=5,time=slice(0,20))
print(xds)

# References to the corrected correlations
RR_corr=np.ravel(xds.CORRECTED_DATA[:,:,0])
RL_corr=np.ravel(xds.CORRECTED_DATA[:,:,1])
LR_corr=np.ravel(xds.CORRECTED_DATA[:,:,2])
LL_corr=np.ravel(xds.CORRECTED_DATA[:,:,3])

# References to the raw data correlations
RR_data=np.ravel(xds.DATA[:,:,0])
RL_data=np.ravel(xds.DATA[:,:,1])
LR_data=np.ravel(xds.DATA[:,:,2])
```

(continues on next page)

(continued from previous page)

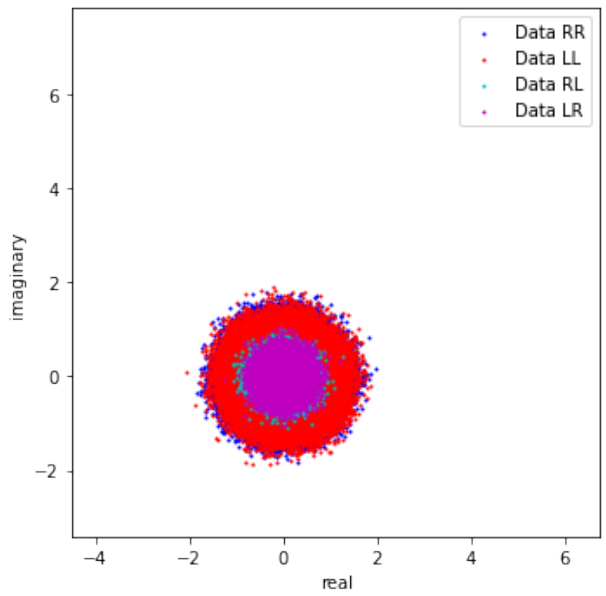
```
LL_data=np.ravel(xds.DATA[:,:,3])


plt.figure(figsize=(12,8))
plt.subplot(122)
plt.scatter(RR_corr.real,RR_corr.imag,5,label='Corrected RR',c='b',marker='.')
plt.scatter(LL_corr.real,LL_corr.imag,5,label='Corrected LL',c='r',marker='.')
plt.scatter(RL_corr.real,RL_corr.imag,5,label='Corrected RL',c='c',marker='.')
plt.scatter(LR_corr.real,LR_corr.imag,5,label='Corrected LR',c='m',marker='.')
plt.xlabel('real')
plt.ylabel('imag')
plt.legend()
ax=list(plt.axis('square'))

plt.subplot(121)
plt.scatter(RR_data.real,RR_data.imag,5,label='Data RR',c='b',marker='.')
plt.scatter(LL_data.real,LL_data.imag,5,label='Data LL',c='r',marker='.')
plt.scatter(RL_data.real,RL_data.imag,5,label='Data RL',c='c',marker='.')
plt.scatter(LR_data.real,LR_data.imag,5,label='Data LR',c='m',marker='.')
plt.xlabel('real')
plt.ylabel('imaginary')
plt.legend()
plt.axis('square')
plt.axis(ax)

plt.show()
```

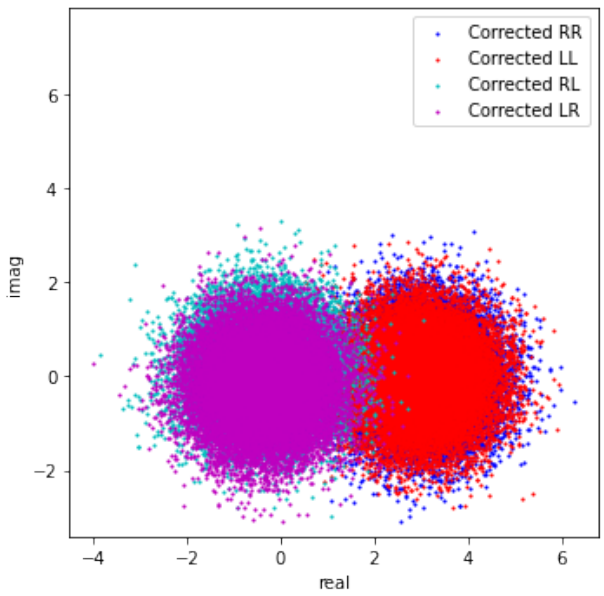
```
overwrite_encoded_chunks True
<xarray.Dataset>
Dimensions:                (baseline: 1128, flag_info: 3, pol: 4, pol_id: 1, spw_id:␣
↪1, time: 20, uvw_index: 3)
Coordinates:
  * baseline               (baseline) int64 0 1 2 3 4 ... 1124 1125 1126 1127
    chan                   float64 1.38e+11
    chan_width             float64 dask.array<chunksize=(), meta=np.ndarray>
    effective_bw           float64 dask.array<chunksize=(), meta=np.ndarray>
  * pol                    (pol) int32 5 6 7 8
  * pol_id                 (pol_id) int32 2
    resolution             float64 dask.array<chunksize=(), meta=np.ndarray>
  * spw_id                 (spw_id) int32 0
  * time                   (time) datetime64[ns] 2019-01-22T20:06:24.67199993...
Dimensions without coordinates: flag_info, uvw_index
Data variables: (12/22)
    ANTENNA1               (baseline) int32 dask.array<chunksize=(1128,), meta=np.
↪ndarray>
    ANTENNA2               (baseline) int32 dask.array<chunksize=(1128,), meta=np.
↪ndarray>
    ARRAY_ID               (time, baseline) int32 dask.array<chunksize=(20, 1128),␣
↪meta=np.ndarray>
    CORRECTED_DATA         (time, baseline, pol) complex128 dask.array<chunksize=(20,␣
↪1128, 4), meta=np.ndarray>
    CORRECTED_DATA_WEIGHT  (time, baseline, pol) complex128 dask.array<chunksize=(20,␣
↪1128, 4), meta=np.ndarray>
    CORRECTED_FLAG         (time, baseline, pol) complex128 dask.array<chunksize=(20,␣
↪1128, 4), meta=np.ndarray>
    ...                     ...
```

(continues on next page)

(continued from previous page)

```
    OBSERVATION_ID          (time, baseline) int32 dask.array<chunksize=(20, 1128),␣
↪meta=np.ndarray>
    PROCESSOR_ID            (time, baseline) int32 dask.array<chunksize=(20, 1128),␣
↪meta=np.ndarray>
    SCAN_NUMBER             (time, baseline) int32 dask.array<chunksize=(20, 1128),␣
↪meta=np.ndarray>
    STATE_ID                (time, baseline) int32 dask.array<chunksize=(20, 1128),␣
↪meta=np.ndarray>
    TIME_CENTROID           (time, baseline) float64 dask.array<chunksize=(20, 1128),␣
↪meta=np.ndarray>
    UVW                     (time, baseline, uvw_index) float64 dask.array
↪<chunksize=(20, 1128, 3), meta=np.ndarray>
Attributes: (12/16)
    bbc_no:                 1
    corr_product:           [[0, 0], [0, 1], [1, 0], [1, 1]]
    data_groups:            [{'0': {'corrected_data': 'CORRECTED_DATA', 'correc...
    freq_group:             0
    freq_group_name:
    if_conv_chain:          0
    ...                     ...
    num_corr:               4
    ref_frequency:          137885200000.0
    sdm_num_bin:            1
    sdm_window_function:    HANNING
    total_bandwidth:        343750000.0
    write_zarr_time:        93.75184869766235
```





6.4 Plot results II

In this section, we use CNGI functions to form a dataset averaged over time and frequency, and this better reveals the quality of the fully-calibrated result (as well as the nature of the systematic gain errors in the un-calibrated data). In the plot, the uncalibrated data are distributed in phase and amplitude. After calibration, the visibilities tightly cluster around the expected values: Stokes I at ~3 Jy (zero phase), Stokes V ~zero, and the cross-hands (conjugates of each other) revealing Q~-0.36 and U~0.15.

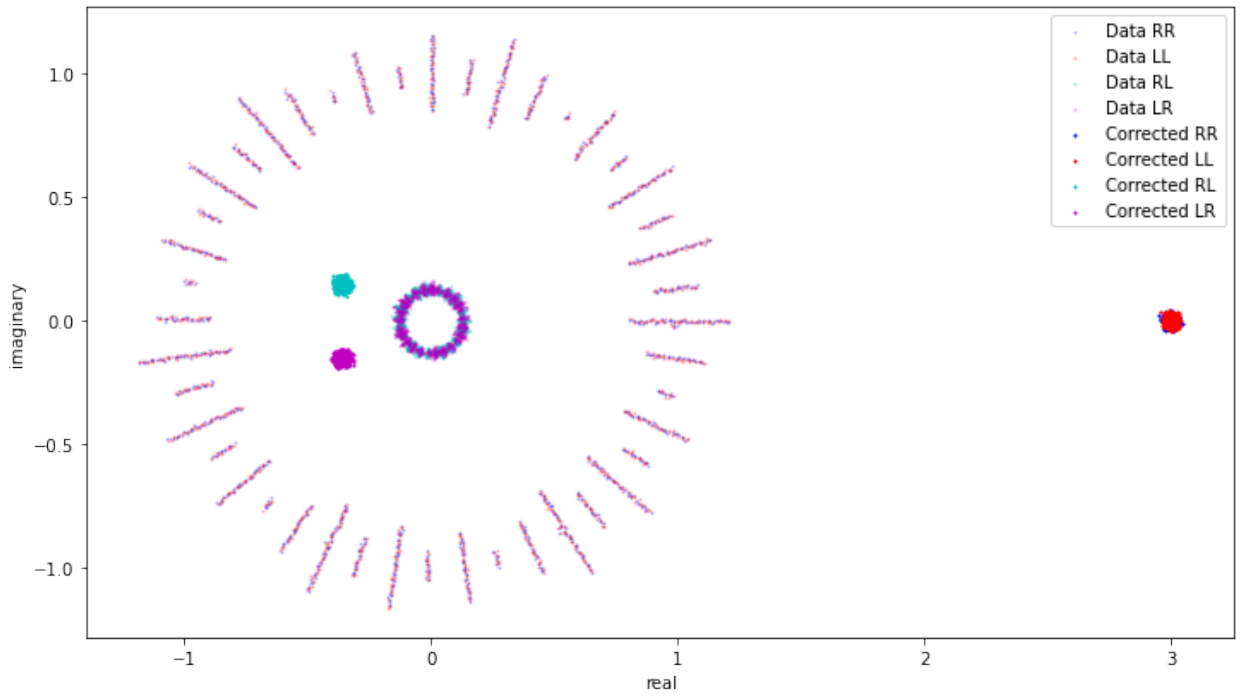
[4]:
```
import xarray as xr
import matplotlib.pylab as plt
import numpy as np
from cngi.vis import chan_average, time_average
from cngi.dio import read_vis
import dask

# Import the calibrated data
sc_vis_mxds = read_vis("cor_selfcal_sim1_1scan.vis.zarr",chunks={'time':150})

# Average in time and frequency via cngi function
mxds = time_average(chan_average(sc_vis_mxds,'xds0',width=11),'xds0',bin=300)
#print(mxds.xds0)
xds = mxds.xds0

# Average over baseline and report the calibrated Stokes parameters
CORR=np.mean(xds.CORRECTED_DATA,(0,1,2)).compute()
I=np.real(CORR[0]+CORR[3])/2.0
Q=np.real(CORR[1]+CORR[2])/2.0
U=np.imag(CORR[1]-CORR[2])/2.0
V=np.real(CORR[0]-CORR[3])/2.0
print('I=',I)
print('Q=',Q)
print('U=',U)
print('V=',V)

# References to the corrected correlations
RR_corr=np.ravel(xds.CORRECTED_DATA[:,:,:,0])
RL_corr=np.ravel(xds.CORRECTED_DATA[:,:,:,1])
LR_corr=np.ravel(xds.CORRECTED_DATA[:,:,:,2])
LL_corr=np.ravel(xds.CORRECTED_DATA[:,:,:,3])

# References to the raw data correlations
RR_data=np.ravel(xds.DATA[:,:,:,0])
RL_data=np.ravel(xds.DATA[:,:,:,1])
LR_data=np.ravel(xds.DATA[:,:,:,2])
LL_data=np.ravel(xds.DATA[:,:,:,3])

# Assemble the plot
plt.figure(figsize=(12,12))

# . . plot the uncorrected data
plt.scatter(RR_data.real,RR_data.imag,5,label='Data RR',c='b',marker='.',alpha=0.2)
plt.scatter(LL_data.real,LL_data.imag,5,label='Data LL',c='r',marker='.',alpha=0.2)
plt.scatter(RL_data.real,RL_data.imag,5,label='Data RL',c='c',marker='.',alpha=0.2)
plt.scatter(LR_data.real,LR_data.imag,5,label='Data LR',c='m',marker='.',alpha=0.2)

# . . plot the corrected data
plt.scatter(RR_corr.real,RR_corr.imag,5,label='Corrected RR',c='b',marker='.')
```

(continues on next page)

(continued from previous page)

```
plt.scatter(LL_corr.real,LL_corr.imag,5,label='Corrected LL',c='r',marker='.')
plt.scatter(RL_corr.real,RL_corr.imag,5,label='Corrected RL',c='c',marker='.')
plt.scatter(LR_corr.real,LR_corr.imag,5,label='Corrected LR',c='m',marker='.')

# labels, etc.
plt.xlabel('real')
plt.ylabel('imaginary')
plt.legend()
plt.axis('scaled')
plt.show()
```

```
overwrite_encoded_chunks True
I= <xarray.DataArray 'CORRECTED_DATA' ()>
array(3.00001346)
Q= <xarray.DataArray 'CORRECTED_DATA' ()>
array(-0.35983528)
U= <xarray.DataArray 'CORRECTED_DATA' ()>
array(0.14953174)
V= <xarray.DataArray 'CORRECTED_DATA' ()>
array(-0.00058128)
```



Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/flagging.ipynb

CHAPTER

SEVEN

FLAGGING

Demonstration of prototype ngCASA flagging functions built from the CNGI infrastructure.

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here

7.1 Installation

[1]:
```
import os
print("installing casa6 + cngi (takes a minute or two)...")
os.system("apt-get install libgfortran3")
os.system("pip install casatasks==6.3.0.48")
os.system("pip install casadata")
os.system("pip install cngi-prototype==0.0.91")

# Retrieve and extract demonstration datasets
print('retrieving MS tarfiles...')
!gdown -q --id 1N9QSs2Hbhi-BrEHx5PA54WigXt8GGgx1
!tar -xzf sis14_twhya_calibrated_flagged.ms.tar.gz
print('complete')
```

```
installing casa6 + cngi (takes a minute or two)...
retrieving MS tarfiles...
complete
```

7.2 Convert MeasurementSet

[2]:
```
from cngi.conversion import convert_ms

zarr_name = 'twhya.vis.zarr'
mxds = convert_ms('sis14_twhya_calibrated_flagged.ms', outfile=zarr_name)
```

```
Completed ddi 0  process time 24.66 s
Completed subtables  process time 1.04 s
```

[3]:
```
from cngi.dio import describe_vis

# We will be working with the only SPW (xds0) present in this dataset, for simplicity
describe_vis(zarr_name)
```

[3]:
```
      spw_id  pol_id  times  baselines  chans  pols  size_MB
xds
xds0       0       0    410        210    384     2     1586
```

7.3 Flag Summaries

Summaries of flags by different dimensions in a dictionary, similar to the dictionary produce by CASA flagdata in 'summary' mode.

[4]:
```
from ngcasa.flagging import summary

# <temporary> prepare for summaries
mxds.xds0['presence_baseline'] = mxds.xds0.DATA.notnull().any(['chan', 'pol'])
# </temporary>

# Get the initial flagging status
counts = summary(mxds, 0)
counts
```

[4]:
```
{'antenna': {'DA42': {'flagged': 4560, 'total': 5763072},
  'DA44': {'flagged': 4560, 'total': 6081792},
  'DA45': {'flagged': 4560, 'total': 6067200},
  'DA46': {'flagged': 4560, 'total': 6067200},
  'DA48': {'flagged': 4560, 'total': 6067200},
  'DA49': {'flagged': 4560, 'total': 6067200},
  'DA50': {'flagged': 4560, 'total': 6067200},
  'DV02': {'flagged': 4560, 'total': 6040320},
  'DV05': {'flagged': 4560, 'total': 6040320},
  'DV06': {'flagged': 4560, 'total': 6081792},
  'DV08': {'flagged': 4560, 'total': 6068736},
  'DV10': {'flagged': 4560, 'total': 6054144},
  'DV13': {'flagged': 4560, 'total': 6039552},
  'DV15': {'flagged': 4560, 'total': 6053376},
  'DV16': {'flagged': 4560, 'total': 6081792},
  'DV17': {'flagged': 4560, 'total': 6039552},
  'DV18': {'flagged': 4560, 'total': 6067200},
  'DV19': {'flagged': 0, 'total': 4808448},
  'DV20': {'flagged': 4560, 'total': 4757760},
  'DV22': {'flagged': 4560, 'total': 6053376},
  'DV23': {'flagged': 4560, 'total': 5377536}},
 'array': {'0': {'flagged': 45600, 'total': 61872384}},
 'correlation': {'XX': {'flagged': 22800, 'total': 30936192},
  'YY': {'flagged': 22800, 'total': 30936192}},
 'field': {'3c279': {'flagged': 0, 'total': 2612736},
  'Ceres': {'flagged': 45600, 'total': 2918400},
  'J0522-364': {'flagged': 0, 'total': 3225600},
  'J1037-295': {'flagged': 0, 'total': 12288000},
  'TW Hya': {'flagged': 0, 'total': 40827648}},
 'flagged': 45600,
 'observation': {'0': {'flagged': 45600, 'total': 61872384}},
 'scan': {'10': {'flagged': 0, 'total': 1459200},
  '12': {'flagged': 0, 'total': 6538752},
  '14': {'flagged': 0, 'total': 1459200},
  '16': {'flagged': 0, 'total': 7956480},
```

(continues on next page)

(continued from previous page)

```
  '18': {'flagged': 0, 'total': 1612800},
  '20': {'flagged': 0, 'total': 7926528},
  '22': {'flagged': 0, 'total': 1612800},
  '24': {'flagged': 0, 'total': 7928832},
  '26': {'flagged': 0, 'total': 1612800},
  '28': {'flagged': 0, 'total': 7266816},
  '30': {'flagged': 0, 'total': 1459200},
  '33': {'flagged': 0, 'total': 2612736},
  '34': {'flagged': 0, 'total': 1459200},
  '36': {'flagged': 0, 'total': 3210240},
  '38': {'flagged': 0, 'total': 1612800},
  '4': {'flagged': 0, 'total': 3225600},
  '7': {'flagged': 45600, 'total': 2918400}},
 'total': 61872384}
```

7.4 Flag Versions

[5]:
```
from ngcasa.flagging import manager_list, manager_add, manager_remove

print('* Printing list of flag variables - checkpoint 0:')
versions = manager_list(mxds.xds0)
print(versions)

vis_flags = manager_add(mxds.xds0, 'FLAG_START', 'flags state at start')
vis_flags = manager_add(vis_flags, 'FLAG_BACKUP', 'backup description')
vis_flags = manager_add(vis_flags, 'FLAG_FINAL', 'backup second descr')

print('* Printing list - checkpoint A:')
versions = manager_list(vis_flags)
print(versions)

vis_flags['FLAG'] = vis_flags['FLAG_BACKUP'] | vis_flags['FLAG_START']

# We can always drop versions that are no longer useful
vis_flags = manager_remove(vis_flags, 'FLAG_START')
vis_flags = manager_remove(vis_flags, 'FLAG_BACKUP')
vis_flags = manager_remove(vis_flags, 'FLAG_FINAL')

print('* Printing list - checkpoint B:')
versions = manager_list(vis_flags)
print(versions)

# FLAG variables as added as regular data variables in the xarray Datasets
# An aditional attribute (flag_variables) are added for bookkeeping
print(vis_flags)
```

```
* Printing list of flag variables - checkpoint 0:
  Flag variable name             Description
0               FLAG  Default flags variable
* Printing list - checkpoint A:
  Flag variable name             Description
0               FLAG  Default flags variable
1         FLAG_START    flags state at start
2        FLAG_BACKUP      backup description
```

(continues on next page)

(continued from previous page)

```
3         FLAG_FINAL     backup second descr
* Printing list - checkpoint B:
  Flag variable name             Description
0               FLAG  Default flags variable
<xarray.Dataset>
Dimensions:            (baseline: 210, chan: 384, pol: 2, pol_id: 1, spw_id: 1, time:␣
↪410, uvw_index: 3)
Coordinates:
  * baseline           (baseline) int64 0 1 2 3 4 5 ... 204 205 206 207 208 209
  * chan               (chan) float64 3.725e+11 3.725e+11 ... 3.728e+11
    chan_width         (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
    effective_bw       (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
  * pol                (pol) int64 9 12
  * pol_id             (pol_id) int64 0
    resolution         (chan) float64 dask.array<chunksize=(32,), meta=np.ndarray>
  * spw_id             (spw_id) int64 0
  * time               (time) datetime64[ns] 2012-11-19T07:37:00 ... 2012-11-...
Dimensions without coordinates: uvw_index
Data variables: (12/21)
    ANTENNA1           (baseline) int64 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2           (baseline) int64 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID           (time, baseline) int64 dask.array<chunksize=(100, 210),␣
↪meta=np.ndarray>
    DATA               (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(100, 210, 32, 1), meta=np.ndarray>
    DATA_WEIGHT        (time, baseline, chan, pol) float64 dask.array<chunksize=(100,␣
↪210, 32, 1), meta=np.ndarray>
    EXPOSURE           (time, baseline) float64 dask.array<chunksize=(100, 210),␣
↪meta=np.ndarray>
    ...                 ...
    TIME_CENTROID      (time, baseline) float64 dask.array<chunksize=(100, 210),␣
↪meta=np.ndarray>
    UVW                (time, baseline, uvw_index) float64 dask.array<chunksize=(100,␣
↪210, 3), meta=np.ndarray>
    presence_baseline  (time, baseline) bool dask.array<chunksize=(100, 210), meta=np.
↪ndarray>
    FLAG_START         (time, baseline, chan, pol) bool dask.array<chunksize=(100,␣
↪210, 32, 1), meta=np.ndarray>
    FLAG_BACKUP        (time, baseline, chan, pol) bool dask.array<chunksize=(100,␣
↪210, 32, 1), meta=np.ndarray>
    FLAG_FINAL         (time, baseline, chan, pol) bool dask.array<chunksize=(100,␣
↪210, 32, 1), meta=np.ndarray>
Attributes: (12/15)
    assoc_nature:     ['', '', '', '', '', '', '', '', '', '', '', '', '', ''...
    bbc_no:           2
    corr_product:     [[0, 0], [1, 1]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    ...               ...
    net_sideband:     2
    num_chan:         384
    num_corr:         2
    ref_frequency:    372533086425.9812
    total_bandwidth:  234375000.0
    flag_variables:   {'FLAG': 'Default flags variable'}
```

7.5 Running flagging methods

A few illustrative examples, trying to cover all the categories of flagging methods.

7.5.1 Meta-Information Based Methods

Methods based on data selection and/or meta-information. Simple examples with data selection based flagging and unflagging.

[6]:
```
from ngcasa.flagging import manual_flag, manual_unflag

vis_dset = manager_add(mxds.xds0, 'FLAG_CHECKPOINT_A', 'after a couple of example␣
↪manual selections')
vis_dset = manager_add(vis_dset, 'FLAG_MANUAL_SELS', 'after applying a few manual␣
↪flags')

# unflag one antenna (not present)
vis_partial = manual_unflag(mxds, 0, [{'antenna': 'DV06'}])

# unflag one antenna (present)
vis_partial = manual_unflag(mxds, 0, [{'antenna': 'DV02'}])

# unflag all
vis_unflagged = manual_unflag(mxds, 0, [])

# <tmp> Handle return xds / mxds(use cngi._utils._io.vis_xds_packager?)
mxds_manual = mxds
mxds_manual.xds0['FLAG'] = vis_unflagged['FLAG']
# </tmp>

counts = summary(mxds_manual, 0)
print('\n* Flags after unflagging:')
counts
```

```
* Flags after unflagging:
```

[6]:
```
{'antenna': {'DA42': {'flagged': 0, 'total': 5763072},
  'DA44': {'flagged': 0, 'total': 6081792},
  'DA45': {'flagged': 0, 'total': 6067200},
  'DA46': {'flagged': 0, 'total': 6067200},
  'DA48': {'flagged': 0, 'total': 6067200},
  'DA49': {'flagged': 0, 'total': 6067200},
  'DA50': {'flagged': 0, 'total': 6067200},
  'DV02': {'flagged': 0, 'total': 6040320},
  'DV05': {'flagged': 0, 'total': 6040320},
  'DV06': {'flagged': 0, 'total': 6081792},
  'DV08': {'flagged': 0, 'total': 6068736},
  'DV10': {'flagged': 0, 'total': 6054144},
  'DV13': {'flagged': 0, 'total': 6039552},
  'DV15': {'flagged': 0, 'total': 6053376},
  'DV16': {'flagged': 0, 'total': 6081792},
  'DV17': {'flagged': 0, 'total': 6039552},
  'DV18': {'flagged': 0, 'total': 6067200},
  'DV19': {'flagged': 0, 'total': 4808448},
  'DV20': {'flagged': 0, 'total': 4757760},
  'DV22': {'flagged': 0, 'total': 6053376},
```

(continues on next page)

(continued from previous page)

```
 'DV23': {'flagged': 0, 'total': 5377536}},
'array': {'0': {'flagged': 0, 'total': 61872384}},
'correlation': {'XX': {'flagged': 0, 'total': 30936192},
 'YY': {'flagged': 0, 'total': 30936192}},
'field': {'3c279': {'flagged': 0, 'total': 2612736},
 'Ceres': {'flagged': 0, 'total': 2918400},
 'J0522-364': {'flagged': 0, 'total': 3225600},
 'J1037-295': {'flagged': 0, 'total': 12288000},
 'TW Hya': {'flagged': 0, 'total': 40827648}},
'flagged': 0,
'observation': {'0': {'flagged': 0, 'total': 61872384}},
'scan': {'10': {'flagged': 0, 'total': 1459200},
 '12': {'flagged': 0, 'total': 6538752},
 '14': {'flagged': 0, 'total': 1459200},
 '16': {'flagged': 0, 'total': 7956480},
 '18': {'flagged': 0, 'total': 1612800},
 '20': {'flagged': 0, 'total': 7926528},
 '22': {'flagged': 0, 'total': 1612800},
 '24': {'flagged': 0, 'total': 7928832},
 '26': {'flagged': 0, 'total': 1612800},
 '28': {'flagged': 0, 'total': 7266816},
 '30': {'flagged': 0, 'total': 1459200},
 '33': {'flagged': 0, 'total': 2612736},
 '34': {'flagged': 0, 'total': 1459200},
 '36': {'flagged': 0, 'total': 3210240},
 '38': {'flagged': 0, 'total': 1612800},
 '4': {'flagged': 0, 'total': 3225600},
 '7': {'flagged': 0, 'total': 2918400}},
'total': 61872384}
```

7.5.2 Command Lists

An example of application of a list of manual flagging commands, resembling use cases from pipelines. An additional required input is the file of flagging commands. Here we use a `.flagonline.txt` file as used in pipelines, where we usually find of the order of 1000s or 10s of thousands of commands. The `.flagonline.txt` is the lion's share of the full list of commands used by pipelines (`.flagcmds.txt`), where the `.flagonline.txt` list of commands is extended with a much shorter list of additional commands that may include a number of summaries, selections based on intent and frequency, the shadow method, etc.

The selection syntax used is the Xarray selection syntax (see examples in the *Visibilities walkthrough*), with selection by label `xds.sel(...)`.

[7]:
```
# Some examples with selections by time, chan, baseline, pol

vis_flags_tr = manual_unflag(mxds, 0, [{'time': slice('2011-09-16T15:38:17','2011-10-
↪16T18:39:50')}])
vis_flags = manual_unflag(mxds, 0, [{}])

# Flag time range
vis_flags_tr = manual_flag(mxds, 0, [{'time': slice('2012-11-19T08:44:55.000', '2012-
↪11-19T08:52:55.400')}])
# Flag antenna in time range
vis_flags_ta = manual_flag(mxds, 0, [{'time': slice('2012-11-19T07:37:00.000', '2012-
↪11-19T08:23:52.800'), 'antenna': 'DA46'}])
```

(continues on next page)

(continued from previous page)

```
# Flag two groups of adjacent ~20 chans
vis_flags_chan = manual_flag(mxds, 0, [{'chan': slice(3.7266e11, 3.7271e11)},
                                       {'chan': slice(3.7276e11, 3.728e11)}])

# Flag polarization, by ID
vis_flags_pol = manual_flag(mxds, 0, [{'pol': 9}])

# Flag some baselines, by ID
vis_flags_base = manual_flag(mxds, 0, [{'baseline': [133, 134, 135]}])

vis_flags_ta['FLAG'] |= vis_flags_tr['FLAG'] | vis_flags_chan['FLAG']
vis_flags_manual = manager_add(vis_flags_ta, 'FLAG_MANUAL_LIST', 'after applying list␣
↪of selection commands', 'FLAG')

# <tmp> Handle return xds / mxds(use cngi._utils._io.vis_xds_packager?)
mxds_manual.xds0['FLAG'] = vis_flags_manual['FLAG']
# </tmp>

print('* Printing list - after flagging:')
versions = manager_list(vis_flags_ta)
print(versions)

print('* Flags after flagging some channels and baselines:')
counts = summary(mxds, 0)
counts
```

```
WARNING: selection results in 0 shape. Sel: {'time': slice(numpy.datetime64('2011-09-
↪16T15:38:17'), numpy.datetime64('2011-10-16T18:39:50'), None)}. Shape: (0, 210, 384,
↪ 2)
* Printing list - after flagging:
  Flag variable name                                Description
0               FLAG                      Default flags variable
1  FLAG_CHECKPOINT_A  after a couple of example manual selections
2   FLAG_MANUAL_SELS            after applying a few manual flags
3   FLAG_MANUAL_LIST    after applying list of selection commands
* Flags after flagging some channels and baselines:
```

[7]:
```
{'antenna': {'DA42': {'flagged': 2067312, 'total': 5763072},
  'DA44': {'flagged': 2145912, 'total': 6081792},
  'DA45': {'flagged': 2142340, 'total': 6067200},
  'DA46': {'flagged': 4302260, 'total': 6067200},
  'DA48': {'flagged': 2141760, 'total': 6067200},
  'DA49': {'flagged': 2142340, 'total': 6067200},
  'DA50': {'flagged': 2141760, 'total': 6067200},
  'DV02': {'flagged': 2135180, 'total': 6040320},
  'DV05': {'flagged': 2124160, 'total': 6040320},
  'DV06': {'flagged': 2145912, 'total': 6081792},
  'DV08': {'flagged': 2142716, 'total': 6068736},
  'DV10': {'flagged': 2138564, 'total': 6054144},
  'DV13': {'flagged': 2135572, 'total': 6039552},
  'DV15': {'flagged': 2138376, 'total': 6053376},
  'DV16': {'flagged': 2145912, 'total': 6081792},
  'DV17': {'flagged': 2133832, 'total': 6039552},
  'DV18': {'flagged': 2142340, 'total': 6067200},
  'DV19': {'flagged': 1782008, 'total': 4808448},
  'DV20': {'flagged': 1282400, 'total': 4757760},
  'DV22': {'flagged': 2128516, 'total': 6053376},
  'DV23': {'flagged': 1944516, 'total': 5377536}},
```

(continues on next page)

(continued from previous page)

```
 'array': {'0': {'flagged': 22801844, 'total': 61872384}},
 'correlation': {'XX': {'flagged': 11400922, 'total': 30936192},
  'YY': {'flagged': 11400922, 'total': 30936192}},
 'field': {'3c279': {'flagged': 639576, 'total': 2612736},
  'Ceres': {'flagged': 934800, 'total': 2918400},
  'J0522-364': {'flagged': 1021600, 'total': 3225600},
  'J1037-295': {'flagged': 4336200, 'total': 12288000},
  'TW Hya': {'flagged': 15869668, 'total': 40827648}},
 'flagged': 22801844,
 'observation': {'0': {'flagged': 22801844, 'total': 61872384}},
 'scan': {'10': {'flagged': 467400, 'total': 1459200},
  '12': {'flagged': 2121472, 'total': 6538752},
  '14': {'flagged': 467400, 'total': 1459200},
  '16': {'flagged': 2523620, 'total': 7956480},
  '18': {'flagged': 510800, 'total': 1612800},
  '20': {'flagged': 2333008, 'total': 7926528},
  '22': {'flagged': 394800, 'total': 1612800},
  '24': {'flagged': 1940912, 'total': 7928832},
  '26': {'flagged': 394800, 'total': 1612800},
  '28': {'flagged': 6164816, 'total': 7266816},
  '30': {'flagged': 1349000, 'total': 1459200},
  '33': {'flagged': 639576, 'total': 2612736},
  '34': {'flagged': 357200, 'total': 1459200},
  '36': {'flagged': 785840, 'total': 3210240},
  '38': {'flagged': 394800, 'total': 1612800},
  '4': {'flagged': 1021600, 'total': 3225600},
  '7': {'flagged': 934800, 'total': 2918400}},
 'total': 61872384}
```

7.5.3 Auto-flagging methods

An illustrative example using the `auto_clip` method. Other auto-flagging methods such as tfcrop, rflag, and uvbin are not implemented.

[8]:
```
from ngcasa.flagging import auto_clip

# vis_dset = manager_add(vis_dset, 'auto_clip_test1', 'after applying clip')
versions = manager_list(vis_dset)
print(versions)

vis_clip = auto_clip(vis_dset, 10, 35)

# <tmp> Handle return xds / mxds (use cngi._utils._io.vis_xds_packager?)
mxds.xds0['FLAG'] = vis_clip['FLAG']
# </tmp>

counts_clip = summary(mxds, 0)
counts_clip
```

```
  Flag variable name                                Description
0               FLAG                      Default flags variable
1  FLAG_CHECKPOINT_A  after a couple of example manual selections
2   FLAG_MANUAL_SELS            after applying a few manual flags
3   FLAG_MANUAL_LIST    after applying list of selection commands
```

[8]:

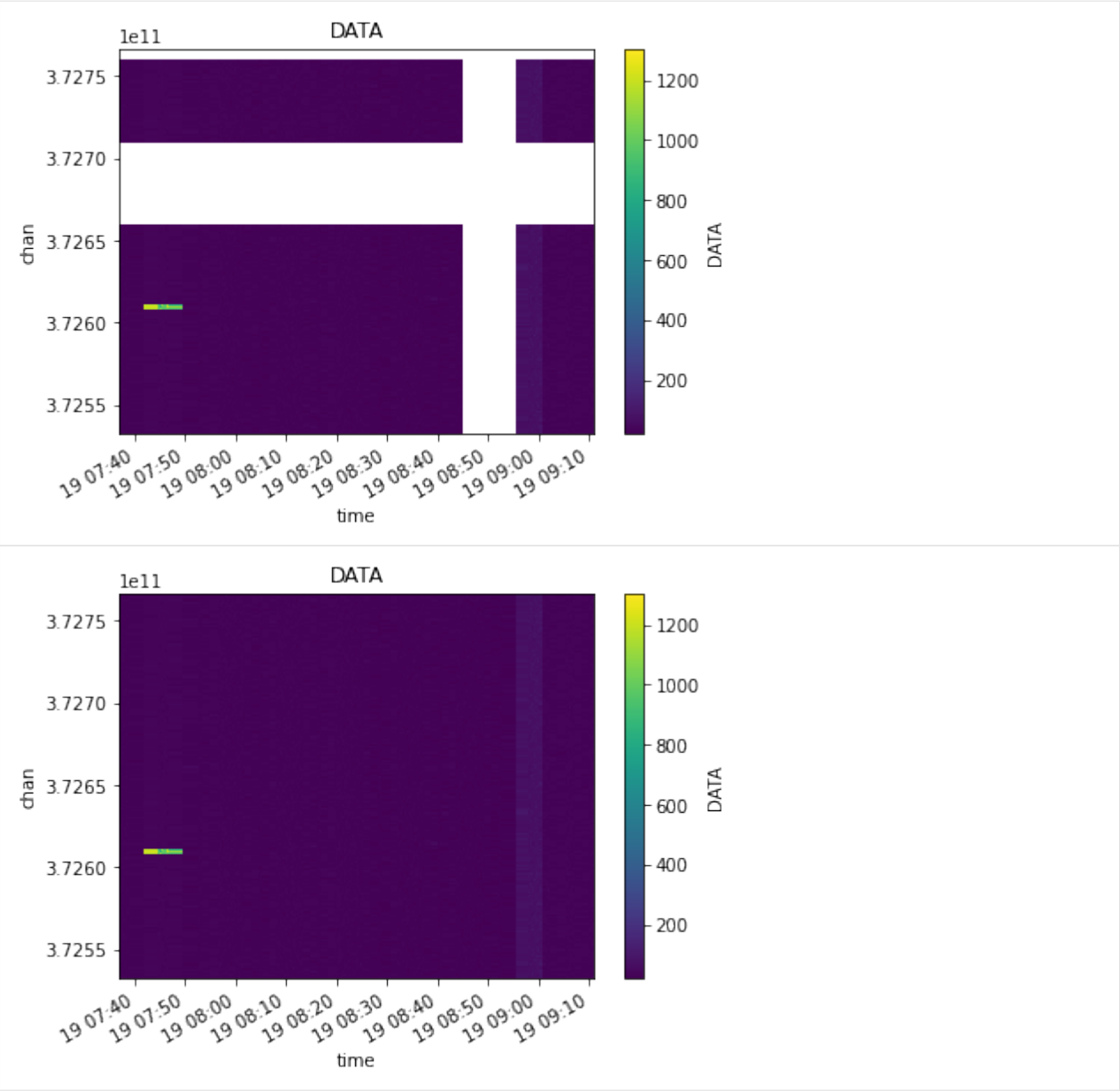
```
{'antenna': {'DA42': {'flagged': 3574131, 'total': 5763072},
  'DA44': {'flagged': 3446333, 'total': 6081792},
  'DA45': {'flagged': 3600687, 'total': 6067200},
  'DA46': {'flagged': 3591839, 'total': 6067200},
  'DA48': {'flagged': 3750212, 'total': 6067200},
  'DA49': {'flagged': 3552130, 'total': 6067200},
  'DA50': {'flagged': 3744628, 'total': 6067200},
  'DV02': {'flagged': 3518095, 'total': 6040320},
  'DV05': {'flagged': 3594624, 'total': 6040320},
  'DV06': {'flagged': 3691423, 'total': 6081792},
  'DV08': {'flagged': 3341354, 'total': 6068736},
  'DV10': {'flagged': 3488830, 'total': 6054144},
  'DV13': {'flagged': 3557439, 'total': 6039552},
  'DV15': {'flagged': 3194677, 'total': 6053376},
  'DV16': {'flagged': 3657900, 'total': 6081792},
  'DV17': {'flagged': 3673673, 'total': 6039552},
  'DV18': {'flagged': 3381264, 'total': 6067200},
  'DV19': {'flagged': 3012167, 'total': 4808448},
  'DV20': {'flagged': 2815375, 'total': 4757760},
  'DV22': {'flagged': 3204976, 'total': 6053376},
  'DV23': {'flagged': 2748667, 'total': 5377536}},
 'array': {'0': {'flagged': 36070212, 'total': 61872384}},
 'correlation': {'XX': {'flagged': 15761725, 'total': 30936192},
  'YY': {'flagged': 20308487, 'total': 30936192}},
 'field': {'3c279': {'flagged': 858487, 'total': 2612736},
  'Ceres': {'flagged': 1375898, 'total': 2918400},
  'J0522-364': {'flagged': 1503426, 'total': 3225600},
  'J1037-295': {'flagged': 7757083, 'total': 12288000},
  'TW Hya': {'flagged': 24575318, 'total': 40827648}},
 'flagged': 36070212,
 'observation': {'0': {'flagged': 36070212, 'total': 61872384}},
 'scan': {'10': {'flagged': 856645, 'total': 1459200},
  '12': {'flagged': 3588925, 'total': 6538752},
  '14': {'flagged': 896040, 'total': 1459200},
  '16': {'flagged': 4885157, 'total': 7956480},
  '18': {'flagged': 1042825, 'total': 1612800},
  '20': {'flagged': 4699547, 'total': 7926528},
  '22': {'flagged': 1051648, 'total': 1612800},
  '24': {'flagged': 4921773, 'total': 7928832},
  '26': {'flagged': 1094965, 'total': 1612800},
  '28': {'flagged': 4555899, 'total': 7266816},
  '30': {'flagged': 970248, 'total': 1459200},
  '33': {'flagged': 858487, 'total': 2612736},
  '34': {'flagged': 867909, 'total': 1459200},
  '36': {'flagged': 1924017, 'total': 3210240},
  '38': {'flagged': 976803, 'total': 1612800},
  '4': {'flagged': 1503426, 'total': 3225600},
  '7': {'flagged': 1375898, 'total': 2918400}},
 'total': 61872384}
```

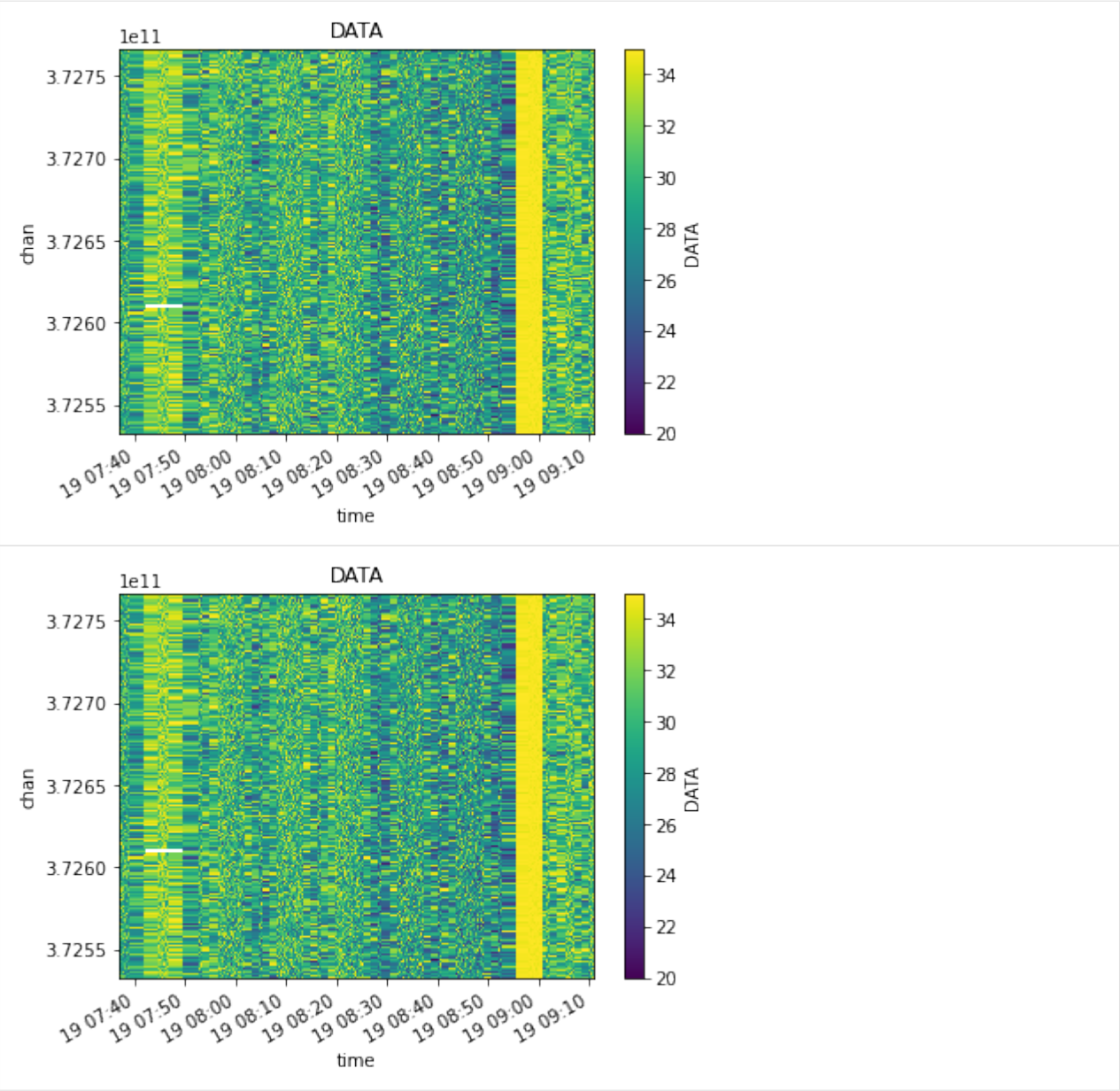
7.6 Applying Flags

To apply a version of flags on a visibilities dataset, before going on to further processing, the function `cngi.vis.apply_flags` should be applied. Some examples can be found in the *Continuum Imaging Example* or the *Visibilities walkthrough*. `cngi.vis.apply_flags` sets the flagged data values to NaN. This has the effect that those NaN values are effectively excluded from subsequent CNGI/ngCASA processing. Other components of CNGI and ngCASA, such as imaging, will ignore those NaN values.

[9]:
```
# Further processing: visualization, calibration, imaging, etc. with flags applied␣
↪(flagged data excluded)
from cngi.vis import apply_flags
from cngi.vis import visplot
from cngi._utils._io import mxds_copier

versions = manager_list(vis_dset)

print(f"* versions of flags: {versions}")
plot_axes = ['time', 'chan']

mxds_manual = mxds.copy()
mxds_manual.attrs['xds0'] = vis_flags_manual

flagged_manual = apply_flags(mxds_manual, 'xds0', flags=['FLAG_MANUAL_LIST'])

visplot(flagged_manual.xds0.DATA, plot_axes)
visplot(mxds.xds0.DATA, plot_axes)

mxds_dset = mxds.copy()
mxds_dset.attrs['xds0'] = vis_dset
mxds_flagged = apply_flags(mxds, 'xds0', flags=['FLAG'])
mxds_clip = mxds.copy()
mxds_clip.attrs['xds0'] = vis_clip
flagged_clip = apply_flags(mxds, 'xds0', flags=['FLAG'])

visplot(mxds_flagged.xds0.DATA, plot_axes)
visplot(flagged_clip.xds0.DATA, plot_axes)
```

```
* versions of flags:   Flag variable name                                  Description
0                FLAG                         Default flags variable
1  FLAG_CHECKPOINT_A  after a couple of example manual selections
2   FLAG_MANUAL_SELS             after applying a few manual flags
3   FLAG_MANUAL_LIST    after applying list of selection commands
```


DATA
1e11
3.7275
3.7270
3.7265
3.7260
3.7255
chan
1200
1000
800
600
400
200
DATA
19 07:40
19 07:50
19 08:00
19 08:10
19 08:20
19 08:30
19 08:40
19 08:50
19 09:00
19 09:10
time
DATA
1e11
3.7275
3.7270
3.7265
3.7260
3.7255
chan
1200
1000
800
600
400
200
DATA
19 07:40
19 07:50
19 08:00
19 08:10
19 08:20
19 08:30
19 08:40
19 08:50
19 09:00
19 09:10
time



DATA
1e11
3.7275
3.7270
3.7265
3.7260
3.7255
chan
19 07:40
19 07:50
19 08:00
19 08:10
19 08:20
19 08:30
19 08:40
19 08:50
19 09:00
19 09:10
time
34
32
30
28
26
24
22
20
DATA

CHAPTER

EIGHT

IMAGING

ngCASA demonstration of imaging prototype functions

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/imaging/continuum_imaging_example.ipynb

8.1 Continuum Imaging

This notebook will demonstrate how to create a continuum dirty image with natural weighting using ngCASA. The resulting image will be compared with an image created by CASA.

For this demonstration data from the ALMA First Look at Imaging CASAguide (https://casaguides.nrao.edu/index.php/First_Look_at_Imaging) will be used. The measurement set has been converted to vis.zarr (using convert_ms in cngi.conversion)

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here.

8.1.1 Installation and Dataset Download

[1]:
```
import os
os.system("pip install cngi-prototype==0.0.91")

!gdown -q --id 1PNL0ANqnN7eyYOpQ_vMslQlorUhtrUmE
!unzip sis14_twhya_field_5_lsrk_pol_xx.vis.zarr.zip > /dev/null

!gdown -q --id 19VlXp9Qyx60ld0rBSAYMVFTq5Y1KuxO8
!unzip casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr.zip > /dev/null

%matplotlib widget
print('complete')
```

```
complete
```

8.1.2 Load Dataset

Two datasets are are needed for this notebook sis14_twhya_field_5_lsrk_pol_xx.vis.zarr and casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr (for more information about the img.zarr format go here and for the vis.zarr format go here).

The sis14_twhya_field_5_lsrk_pol_xx.vis.zarr dataset is used to create a continuum image. The dataset was created by using the `mstransform` command in CASA

```
mstransform('sis14_twhya_calibrated_flagged.ms',
            outputvis='sis14_twhya_field_5_lsrk_pol_xx.ms',
            regridms=True, outframe='LSRK', datacolumn='data',
            correlation='XX',field='5')
```

and then convert_ms in cngi.conversion

```
infile = 'sis14_twhya_field_5_lsrk_pol_xx.ms'
outfile = 'sis14_twhya_field_5_lsrk_pol_xx.vis.zarr'
chunk_shape=(270, 210, 12, 1)
convert_ms(infile, outfile=outfile , chunk_shape=chunk_shape)
```

The conversion to 'LSRK' is necessary because cngi does not currently have an implementation and tclean does a conversion to 'LSRK' before imaging.

To check the ngcasa imaging results the casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr dataset is used. This dataset was generated by running `tclean` in CASA

```
tclean(vis='sis14_twhya_field_5_lsrk_pol_xx.ms',
       imagename='twhya_standard_gridder_lsrk_mfs_natural',
       specmode='mfs',
       deconvolver='hogbom',
       imsize=[200,400],
       cell=['0.08arcsec'],
       weighting='natural',
       threshold='0mJy',
       niter=0,stokes='XX')
```

and then `image_ms` in cngi.conversion

```
infile = 'twhya_standard_gridder_lsrk_mfs_natural.image'
outfile = 'casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr'
convert_image(infile=infile,outfile=outfile)
```

[2]:
```
import xarray as xr
from cngi.dio import read_vis, read_image

xr.set_options(display_style="html")

mxds = read_vis("sis14_twhya_field_5_lsrk_pol_xx.vis.zarr",chunks={'chan':192})
mxds.xds0
```

```
overwrite_encoded_chunks True
```

[2]:
```
<xarray.Dataset>
Dimensions:          (baseline: 210, chan: 384, pol: 1, pol_id: 1, spw_id: 1, time:␣
↪270, uvw_index: 3)
Coordinates:
  * baseline         (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
```

(continues on next page)

(continued from previous page)

```
  * chan             (chan) float64 3.725e+11 3.725e+11 ... 3.728e+11 3.728e+11
    chan_width       (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
    effective_bw     (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * pol              (pol) int32 9
  * pol_id           (pol_id) int32 0
    resolution       (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * spw_id           (spw_id) int32 0
  * time             (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1         (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2         (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID         (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    DATA             (time, baseline, chan, pol) complex128 dask.array<chunksize=(270,␣
↪210, 192, 1), meta=np.ndarray>
    DATA_WEIGHT      (time, baseline, chan, pol) float64 dask.array<chunksize=(270,␣
↪210, 192, 1), meta=np.ndarray>
    EXPOSURE         (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    ...               ...
    OBSERVATION_ID   (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    PROCESSOR_ID     (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    SCAN_NUMBER      (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    STATE_ID         (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    TIME_CENTROID    (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    UVW              (time, baseline, uvw_index) float64 dask.array<chunksize=(270,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         384
    num_corr:         1
    ref_frequency:    372520022603.63745
    total_bandwidth:  234366781.0546875
```

Note that the chunks parameter in cngi and ngcasa functions specifies the size of a chunk and not the number of chunks (in CASA `tclean` chanchunks refers to the number of channel chunks).

The dimensionality of the sis14_twhya_field_5_lsrk_pol_xx.vis.zarr dataset is (time:270,baseline:210,chan:384,pol:1) and a zarr chunk size of (time:270,baseline:210,chan:12,pol:1) was chosen. With the `cngi.dio.read_vis` function the dask chunk size was set to (time:270,baseline:210,chan:192,pol:1). For more information concerning chunking go here.

8.1.3 Flag Data and Create Imaging Weights

The `apply_flags` cngi.vis function sets all values that should be flagged to nan. The ngcasa.imaging code does not internally apply flags but does ignore nan values. apply_flags documentation

The `make_imaging_weight` cngi.imaging function takes the WEIGHT data variables and creates IMAGING_WEIGHT data variable that has dimensions time x baseline x chan x pol (matches the visibility DATA variable). Weighting schemes that are supported include natural, uniform, briggs, briggs_abs. make_imaging_weight documentation

[3]:
```
from cngi.vis import apply_flags
from ngcasa.imaging import make_imaging_weight

mxds = apply_flags(mxds, 'xds0', flags='FLAG')

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0

grid_parms = {}
grid_parms['chan_mode'] = 'continuum'
grid_parms['image_size'] = [200,400]
grid_parms['cell_size'] = [0.08,0.08]

mxds = make_imaging_weight(mxds, imaging_weights_parms, grid_parms, sel_parms)

mxds.xds0
```

```
######################### Start make_imaging_weights #########################
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'IMAGING_WEIGHT'}
Since weighting is natural input weight will be reused as imaging weight.
######################### Created graph for make_imaging_weight ######################
↪###
```

[3]:
```
<xarray.Dataset>
Dimensions:         (baseline: 210, chan: 384, pol: 1, pol_id: 1, spw_id: 1, time:␣
↪270, uvw_index: 3)
Coordinates:
  * baseline        (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
  * chan            (chan) float64 3.725e+11 3.725e+11 ... 3.728e+11 3.728e+11
    chan_width      (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
    effective_bw    (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * pol             (pol) int32 9
  * pol_id          (pol_id) int32 0
    resolution      (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * spw_id          (spw_id) int32 0
  * time            (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1        (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2        (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID        (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
```

(continues on next page)

(continued from previous page)

```
    DATA            (time, baseline, chan, pol) complex128 dask.array<chunksize=(270,␣
↪210, 192, 1), meta=np.ndarray>
    DATA_WEIGHT     (time, baseline, chan, pol) float64 dask.array<chunksize=(270,␣
↪210, 192, 1), meta=np.ndarray>
    EXPOSURE        (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    ...              ...
    OBSERVATION_ID  (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    PROCESSOR_ID    (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    SCAN_NUMBER     (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    STATE_ID        (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    TIME_CENTROID   (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    UVW             (time, baseline, uvw_index) float64 dask.array<chunksize=(270,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         384
    num_corr:         1
    ref_frequency:    372520022603.63745
    total_bandwidth:  234366781.0546875
```

8.1.4 Create Dirty Continuum Image and Primary Beam

The `make_image` cngi.imaging function grids the data (using the prolate spheroidal function as an anti-aliasing filter), fast Fourier transform the gridded data to an image and normalizes the image. The `make_pb` function currently supports rotationally symmetric airy disk primary beams. The `write_zarr` function is now used to trigger a compute (which includes applying the flags, creating the imaging weights and making the image).

make_pb documentation

make_image documentation

[4]:
```
from ngcasa.imaging import make_image
from ngcasa.imaging import make_pb
from cngi.dio import write_image
import dask

grid_parms = {}
grid_parms['chan_mode'] = 'continuum'
grid_parms['image_size'] = [200,400]
grid_parms['cell_size'] = [0.08,0.08]
grid_parms['phase_center'] = mxds.FIELD.PHASE_DIR[0,0,:].data.compute()
```

(continues on next page)

(continued from previous page)

```
vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 0

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 0

img_xds = xr.Dataset() #empty dataset

img_xds = make_image(mxds, img_xds, grid_parms, vis_sel_parms, img_sel_parms)


make_pb_parms = {}
make_pb_parms['function'] = 'casa_airy'
make_pb_parms['list_dish_diameters'] = [10.7]
make_pb_parms['list_blockage_diameters'] = [0.75]

sel_parms = {}
sel_parms['img_description_in_indx'] = 0


img_xds = make_pb(img_xds,make_pb_parms, grid_parms, sel_parms)


img_xds = write_image(img_xds, 'twhya_standard_gridder_lsrk_mfs_natural.img.zarr')
```

```
######################### Start make_image #########################
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '1', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0'}
Setting default data_group_out [' sum_weight ']  to  SUM_WEIGHT
Setting default data_group_out [' image ']  to  IMAGE
######################### Created graph for make_image ########################
######################### Start make_pb ########################
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT', 'image':
↪'IMAGE'}
Setting default data_group_out  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT', 'image':
↪'IMAGE', 'pb': 'PB'}
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
######################### Created graph for make_pb ########################
Time to store and execute graph  write_zarr 7.705185174942017
```

[5]:
```
casa_img_xds = read_image("casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr")
casa_img_xds
```

[5]:
```
<xarray.Dataset>
Dimensions:            (chan: 1, l: 200, m: 400, pol: 1, time: 1)
Coordinates:
  * chan               (chan) float64 3.726e+11
    declination        (l, m) float64 dask.array<chunksize=(200, 400), meta=np.
↪ndarray>
```

(continues on next page)

(continued from previous page)

```
  * l                  (l) float64 3.879e-05 3.84e-05 ... -3.801e-05 -3.84e-05
  * m                  (m) float64 -7.757e-05 -7.718e-05 ... 7.679e-05 7.718e-05
  * pol                (pol) float64 9.0
    right_ascension    (l, m) float64 dask.array<chunksize=(200, 400), meta=np.
↪ndarray>
  * time               (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    IMAGE              (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0        (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR        (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0  (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL              (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    PB                 (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0           (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL           (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0     (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT              (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes: (12/19)
    axisnames:            ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:            ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:           [0.6637904644012451, 0.5050938129425049, -65.900085...
    commonbeam_units:     ['arcsec', 'arcsec', 'deg']
    date_observation:     2012/11/19/07
    direction_reference:  j2000
    ...                   ...
    restoringbeam:        [0.6637904644012451, 0.5050938129425049, -65.900085...
    spectral__reference:  lsrk
    telescope:            alma
    telescope_position:   [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                 Jy/beam
    velocity__type:       radio
```

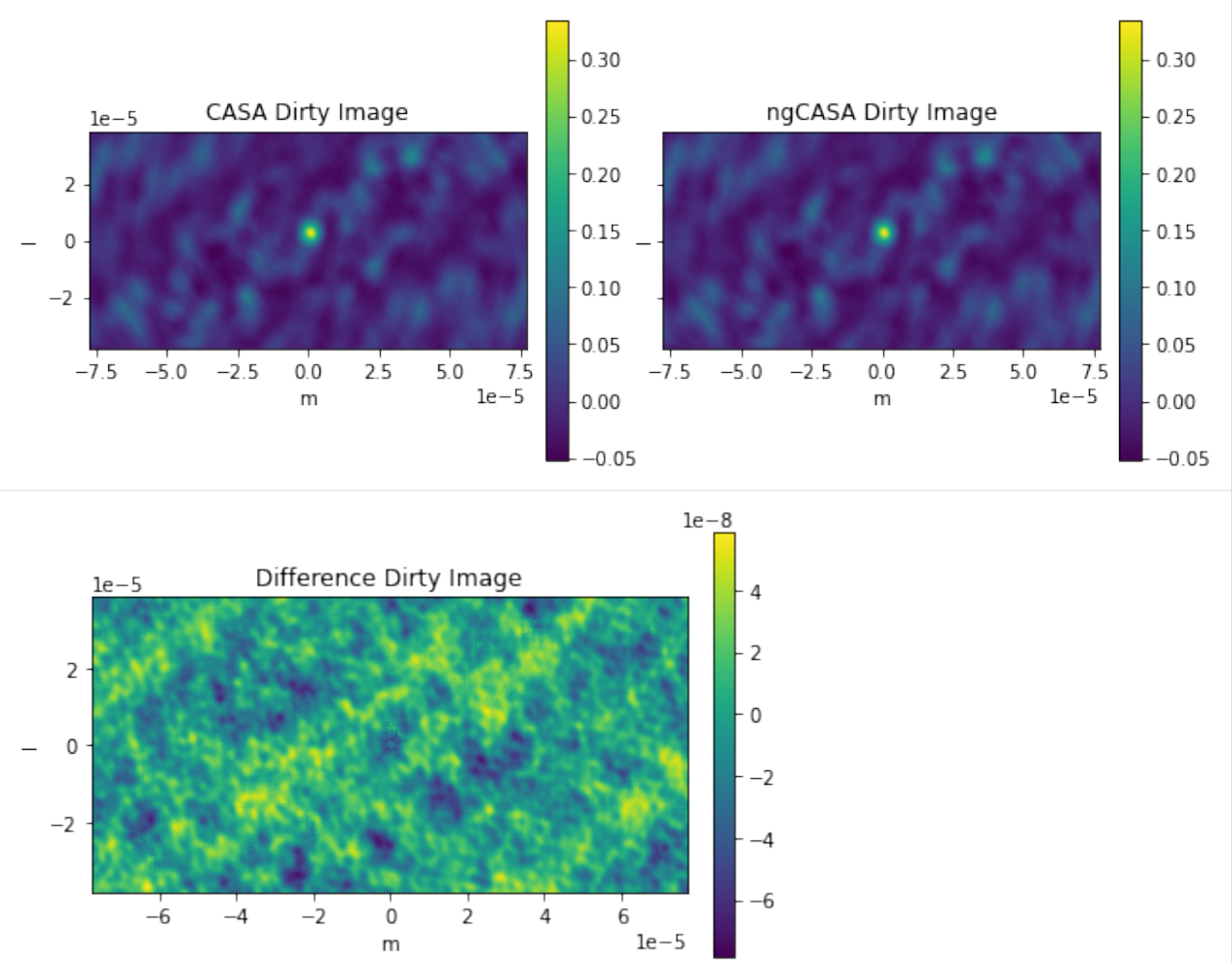
8.1.5 Plot and Compare With CASA

[6]:
```
import matplotlib.pylab as plt
import numpy as np

img_xds = read_image("twhya_standard_gridder_lsrk_mfs_natural.img.zarr")
casa_img_xds = read_image("casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr")
plt.close('all')

#### Compare dirty images ####
dirty_image = img_xds['IMAGE'].isel(chan=0,time=0,pol=0) #Image created by ngCASA
```

(continues on next page)

(continued from previous page)

```
casa_dirty_image = casa_img_xds['RESIDUAL'].isel(chan=0,time=0,pol=0) #Image created␣
↪by CASA

#Plotting Images
minmin = np.min([np.min(casa_dirty_image.data), np.min(dirty_image.data)])
maxmax = np.max([np.max(casa_dirty_image.data), np.max(dirty_image.data)])
extent = extent=(np.min(casa_dirty_image.m),np.max(casa_dirty_image.m),np.min(casa_
↪dirty_image.l),np.max(casa_dirty_image.l))

fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
im0 = ax0[0].imshow(casa_dirty_image,extent=extent)
im1 = ax0[1].imshow(dirty_image,extent=extent)
ax0[0].title.set_text('CASA Dirty Image')
ax0[1].title.set_text('ngCASA Dirty Image')
ax0[0].set_xlabel('m')
ax0[1].set_xlabel('m')
ax0[0].set_ylabel('l')
ax0[1].set_ylabel('l')
fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
plt.show()

dif_image = casa_dirty_image - dirty_image
plt.figure()
plt.imshow(dif_image,extent=extent)
plt.title('Difference Dirty Image')
plt.xlabel('m')
plt.ylabel('l')
plt.colorbar(fraction=0.046, pad=0.04)
plt.show()

dirty_image = dirty_image / np.max(np.abs(dirty_image))
casa_dirty_image = casa_dirty_image / np.max(np.abs(casa_dirty_image))

# Calculate max error
max_error_dirty_image = np.max(np.abs(dirty_image - casa_dirty_image)).values
print('Max Error',max_error_dirty_image)

# Calculate root mean square error
rms_error_dirty_image = np.linalg.norm(dirty_image - casa_dirty_image, 'fro')
print('RMS Error',rms_error_dirty_image)
```




```
Max Error 2.2810074962187343e-07
RMS Error 1.6647486690733482e-05
```

The reason for the small difference between ngCASA and CASA is due to ngCASA using a different implementation of the Fast Fourier Transform.

[7]:
```
import matplotlib.pylab as plt
import numpy as np
import xarray as xr
import dask.array as da
import dask
from cngi.dio import read_image

img_xds = read_image("twhya_standard_gridder_lsrk_mfs_natural.img.zarr").isel(chan=0,␣
↪pol=0, time=0, dish_type=0)
casa_img_xds = read_image("casa_twhya_standard_gridder_lsrk_mfs_natural.img.zarr").
↪isel(chan=0, time=0, pol=0)
plt.close('all')


#### Primary Beam Corrected Images ####
pb_limit = 0.2
```

(continues on next page)

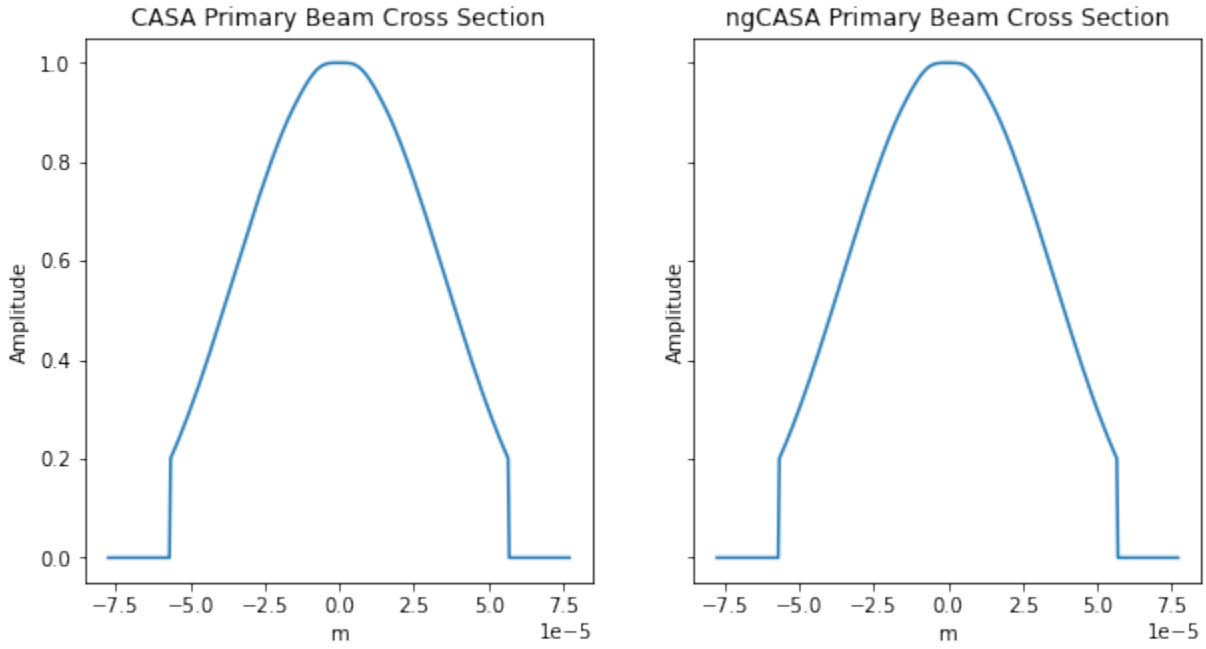
(continued from previous page)

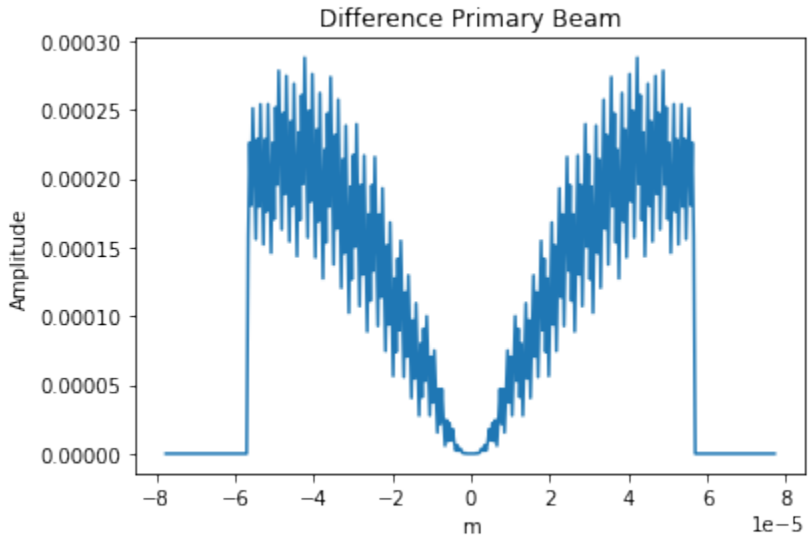
```
primary_beam = img_xds.PB.isel(l=100).where(img_xds.PB.isel(l=100) > pb_limit,other=0.
↪0)
dirty_image_pb_cor = img_xds.IMAGE/img_xds.PB
dirty_image_pb_cor = dirty_image_pb_cor.where(img_xds.PB > pb_limit,other=np.nan)

#print(casa_img_xds)
casa_primary_beam = casa_img_xds['PB'].isel(l=100).data #Primary beam created by CASA
casa_dirty_image_pb_cor = (casa_img_xds['IMAGE_PBCOR']).where(casa_img_xds['PB'] > pb_
↪limit,other=np.nan) #Image created by CASA

#Plot Primary Beams
fig0, ax0, = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
im0 = ax0[0].plot(casa_dirty_image.m,casa_primary_beam)
im1 = ax0[1].plot(casa_dirty_image.m,primary_beam)
ax0[0].title.set_text('CASA Primary Beam Cross Section')
ax0[1].title.set_text('ngCASA Primary Beam Cross Section')
ax0[0].set_xlabel('m')
ax0[1].set_xlabel('m')
ax0[0].set_ylabel('Amplitude')
ax0[1].set_ylabel('Amplitude')
plt.show()

plt.figure()
plt.plot(casa_dirty_image.m,casa_primary_beam-primary_beam)
plt.title('Difference Primary Beam')
plt.xlabel('m')
plt.ylabel('Amplitude')
plt.show()


extent=(np.min(casa_dirty_image.m),np.max(casa_dirty_image.m),np.min(casa_dirty_image.
↪l),np.max(casa_dirty_image.l))

#Plotting Images
fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
im0 = ax0[0].imshow(casa_dirty_image_pb_cor,extent=extent)
im1 = ax0[1].imshow(dirty_image_pb_cor,extent=extent)
ax0[0].title.set_text('CASA PB Corrected Dirty Image')
ax0[1].title.set_text('ngCASA PB Corrected Dirty Image')
ax0[0].set_xlabel('m')
ax0[1].set_xlabel('m')
ax0[0].set_ylabel('l')
ax0[1].set_ylabel('l')
fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
plt.show()

plt.figure()
plt.imshow(casa_dirty_image_pb_cor - dirty_image_pb_cor,extent=extent)
plt.title('Difference Dirty Image')
plt.xlabel('m')
plt.ylabel('l')
plt.colorbar(fraction=0.046, pad=0.04)
plt.show()

dirty_image_pb_cor = dirty_image_pb_cor / np.nanmax(np.abs(dirty_image_pb_cor))
casa_dirty_image_pb_cor = casa_dirty_image_pb_cor / np.nanmax(np.abs(casa_dirty_image_
↪pb_cor))
```

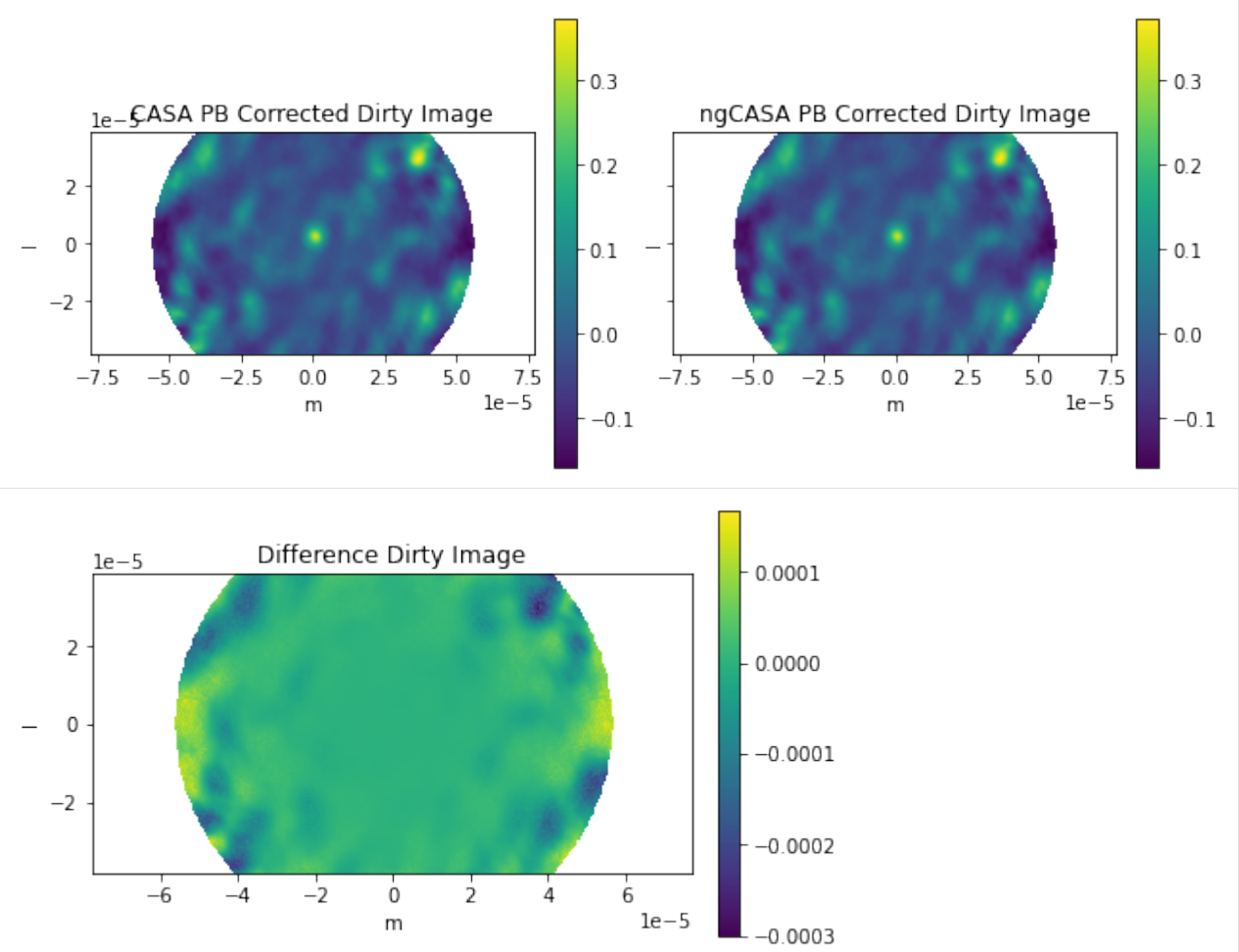
(continues on next page)

(continued from previous page)

```
norm_diff_image_pb_cor = dirty_image_pb_cor - casa_dirty_image_pb_cor

# Calculate max error
max_error_dirty_image = np.nanmax(np.abs(norm_diff_image_pb_cor))
print('Max Normalized Error',max_error_dirty_image)

# Calculate root mean square error
rms_error_dirty_image = np.sqrt(np.nansum(np.square(norm_diff_image_pb_cor)))
print('RMS Normalized Error',rms_error_dirty_image)
```








```
Max Normalized Error 0.0005570343301252167
RMS Normalized Error 0.011389306569008918
```

The difference in primary beam is due to CASA using a sampled 1D function while ngCASA calculates the PB for each pixel. If it is found that PB creation becomes a bottleneck for ngCASA the implementation will be changed to match CASA.

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/imaging/cube_imaging_example.ipynb

---

8.2 Cube Imaging

This notebook will demonstrate how to create a cube dirty image with natural weighting using ngCASA. The resulting image will be compared with an image created by CASA. The Dask visualize tool will also be introduced.

For this demonstration data from the ALMA First Look at Imaging CASAguide (https://casaguides.nrao.edu/index.php/First_Look_at_Imaging) will be used. The measurement set has been converted to vis.zarr (using convert_ms in cngi.conversion).

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here.

8.2.1 Installation and Dataset Download

[1]:
```
import os

os.system("pip install cngi-prototype==0.0.91")

!gdown -q --id 1fN3rdEQoeKExyIVBP87ZHdJTNuugkTJ0
!unzip sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr.zip > /dev/null

!gdown -q --id 1KpMk_BpRVsomnvPRYYAoaVByEhhgg2mU
!unzip casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr.zip > /dev/null

#%matplotlib widget
print('complete')
```

```
complete
```

8.2.2 Load Dataset

Two datasets are are needed for this notebook: - sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr - casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr

(for more information about the img.zarr format go here and for the vis.zarr format go here).

The sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr dataset is used to create a cube image. The dataset was created by using the `mstransform` command in CASA

```
mstransform('sis14_twhya_calibrated_flagged.ms',
            outputvis='sis14_twhya_chan_avg_field_5_lsrk_pol_xx.ms',
            regridms=True, outframe='LSRK', datacolumn='data',
            correlation='XX', field='5', nchan=7)
```

and then convert_ms in cngi.conversion

```
infile = 'sis14_twhya_chan_avg_field_5_lsrk_pol_xx.ms'
outfile = 'sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr'
chunk_shape=(270, 210, 1, 1)
convert_ms(infile, outfile=outfile, chunk_shape=chunk_shape)
```

The conversion to 'LSRK' is necessary because cngi does not currently have an implementation and tclean does a conversion to 'LSRK' before imaging.

To check the ngcasa imaging results the casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr dataset is used. This dataset was generated by running `tclean` in CASA

```
tclean(vis='sis14_twhya_chan_avg_field_5_lsrk_pol_xx.ms',
       imagename='twhya_standard_gridder_lsrk_cube_natural',
       specmode='cube',
       deconvolver='hogbom',
       imsize=[200,400],
       cell=['0.08arcsec'],
       weighting='natural',
       threshold='0mJy',
       niter=0,stokes='XX')
```

and then `image_ms` in cngi.conversion

```
infile = 'cube_image/twhya_standard_gridder_lsrk_cube_natural.image'
outfile = 'casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr'
convert_image(infile=infile,outfile=outfile)
```

[2]:
```
import xarray as xr
from cngi.dio import read_vis, read_image

xr.set_options(display_style="html")

mxds = read_vis("sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr")
print(mxds.xds0)
```

```
overwrite_encoded_chunks True
<xarray.Dataset>
Dimensions:          (baseline: 210, chan: 7, pol: 1, pol_id: 1, spw_id: 1, time: 270,␣
↪uvw_index: 3)
Coordinates:
  * baseline         (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
  * chan             (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11 3.725e+11
    chan_width       (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    effective_bw     (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * pol              (pol) int32 9
  * pol_id           (pol_id) int32 0
    resolution       (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * spw_id           (spw_id) int32 0
  * time             (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1         (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2         (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID         (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    DATA             (time, baseline, chan, pol) complex128 dask.array<chunksize=(270,␣
↪210, 1, 1), meta=np.ndarray>
    DATA_WEIGHT      (time, baseline, chan, pol) float64 dask.array<chunksize=(270,␣
↪210, 1, 1), meta=np.ndarray>
    EXPOSURE         (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    ...               ...
    OBSERVATION_ID   (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    PROCESSOR_ID     (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    SCAN_NUMBER      (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    STATE_ID         (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    TIME_CENTROID    (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    UVW              (time, baseline, uvw_index) float64 dask.array<chunksize=(270,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
```

(continues on next page)

(continued from previous page)

```
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         7
    num_corr:         1
    ref_frequency:    372520022603.63745
    total_bandwidth:  4272311.112915039
```

[3]:
```
casa_img_xds = read_image("casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr")
print(casa_img_xds)
```

```
<xarray.Dataset>
Dimensions:             (chan: 7, l: 200, m: 400, pol: 1, time: 1)
Coordinates:
  * chan                (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11
    declination         (l, m) float64 dask.array<chunksize=(200, 400), meta=np.
↪ndarray>
  * l                   (l) float64 3.879e-05 3.84e-05 ... -3.801e-05 -3.84e-05
  * m                   (m) float64 -7.757e-05 -7.718e-05 ... 7.679e-05 7.718e-05
  * pol                 (pol) float64 9.0
    right_ascension     (l, m) float64 dask.array<chunksize=(200, 400), meta=np.
↪ndarray>
  * time                (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    IMAGE               (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0         (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    IMAGE_PBCOR         (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    IMAGE_PBCOR_MASK0   (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    MODEL               (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    PB                  (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    PB_MASK0            (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    PSF                 (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL            (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 400,
↪ 1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0      (l, m, time, chan, pol) bool dask.array<chunksize=(200, 400, 1,
↪ 1, 1), meta=np.ndarray>
    SUMWT               (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes: (12/19)
    axisnames:            ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:            ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:           [0.6639984846115112, 0.5052574276924133, -65.900550...
    commonbeam_units:     ['arcsec', 'arcsec', 'deg']
    date_observation:     2012/11/19/07
    direction_reference:  j2000
    ...                   ...
    restoringbeam:        [0.6639984846115112, 0.5052574276924133, -65.900550...
    spectral__reference:  lsrk
```

(continues on next page)

(continued from previous page)

```
    telescope:              alma
    telescope_position:     [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                   Jy/beam
    velocity__type:         radio
```

Note that the chunks parameter in cngi and ngcasa functions specifies the size of a chunk and not the number of chunks (in CASA `tclean` chanchunks refers to the number of channel chunks).

The dimensionality of the sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr dataset is (time:270,baseline:210,chan:7,pol:1) and a zarr chunk size of (time:270,baseline:210,chan:1,pol:1) was chosen. The dask chunk size was chosen to be the same as the zarr chunk size. For more information concerning chunking go to here.

8.2.3 Flag Data and Create Imaging Weights

The `apply_flags` cngi.vis function sets all values that should be flagged to nan. The ngcasa.imaging code does not internally apply flags but does ignore nan values. apply_flags documentation

The `make_imaging_weight` cngi.imaging function takes the WEIGHT or WEIGHT_SPECTRUM data variables and creates IMAGING_WEIGHT data variable that has dimensions time x baseline x chan x pol (matches the visibility DATA variable). Weighting schemes that are supported include natural, uniform, briggs, briggs_abs. Using imaging_weights_parms['chan_mode'] = 'cube' is equivalent to perchanweightdensity=True in CASA. make_imaging_weight documentation

[4]:
```
from cngi.vis import apply_flags
from ngcasa.imaging import make_imaging_weight

mxds = apply_flags(mxds, 'xds0', flags='FLAG')

grid_parms = {}
grid_parms['chan_mode'] = 'cube'
grid_parms['image_size'] = [200,400]
grid_parms['cell_size'] = [0.08,0.08]

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'

mxds =  make_imaging_weight(mxds, imaging_weights_parms, grid_parms, sel_parms)

print(mxds.xds0)
```

```
######################### Start make_imaging_weights ########################
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'IMAGING_WEIGHT'}
Since weighting is natural input weight will be reused as imaging weight.
######################### Created graph for make_imaging_weight ######################
↪###
<xarray.Dataset>
Dimensions:         (baseline: 210, chan: 7, pol: 1, pol_id: 1, spw_id: 1, time: 270,␣
↪uvw_index: 3)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * baseline        (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
  * chan            (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11 3.725e+11
    chan_width      (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    effective_bw    (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * pol             (pol) int32 9
  * pol_id          (pol_id) int32 0
    resolution      (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * spw_id          (spw_id) int32 0
  * time            (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1        (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2        (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID        (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    DATA            (time, baseline, chan, pol) complex128 dask.array<chunksize=(270,␣
↪210, 1, 1), meta=np.ndarray>
    DATA_WEIGHT     (time, baseline, chan, pol) float64 dask.array<chunksize=(270,␣
↪210, 1, 1), meta=np.ndarray>
    EXPOSURE        (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    ...              ...
    OBSERVATION_ID  (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    PROCESSOR_ID    (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    SCAN_NUMBER     (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    STATE_ID        (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    TIME_CENTROID   (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    UVW             (time, baseline, uvw_index) float64 dask.array<chunksize=(270,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         7
    num_corr:         1
    ref_frequency:    372520022603.63745
    total_bandwidth:  4272311.112915039
```

8.2.4 Create Dirty Cube Image

The `make_image` cngi.imaging function grids the data (using the prolate spheroidal function as an anti-aliasing filter), fast Fourier transform the gridded data to an image and normalizes the image. The `make_pb` function currently supports rotationally symmetric airy disk primary beams.

make_pb documentation

make_image documentation

To create an image of the execution graph the [dask.visualize]

[5]:
```
from ngcasa.imaging import make_image
from ngcasa.imaging import make_pb
from cngi.dio import write_image
import dask

grid_parms = {}
grid_parms['chan_mode'] = 'cube'
grid_parms['image_size'] = [200,400]
grid_parms['cell_size'] = [0.08,0.08]
grid_parms['phase_center'] = mxds.FIELD.PHASE_DIR[0,0,:].data.compute()

vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 0

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 0

img_xds = xr.Dataset() #empty dataset

img_xds = make_image(mxds, img_xds, grid_parms, vis_sel_parms, img_sel_parms)


make_pb_parms = {}
make_pb_parms['function'] = 'casa_airy'
make_pb_parms['list_dish_diameters'] = [10.7]
make_pb_parms['list_blockage_diameters'] = [0.75]

sel_parms = {}
sel_parms['img_description_in_indx'] = 0


img_xds = make_pb(img_xds,make_pb_parms, grid_parms, sel_parms)
dask.visualize(img_xds,filename='cube_image_graph.png')
```

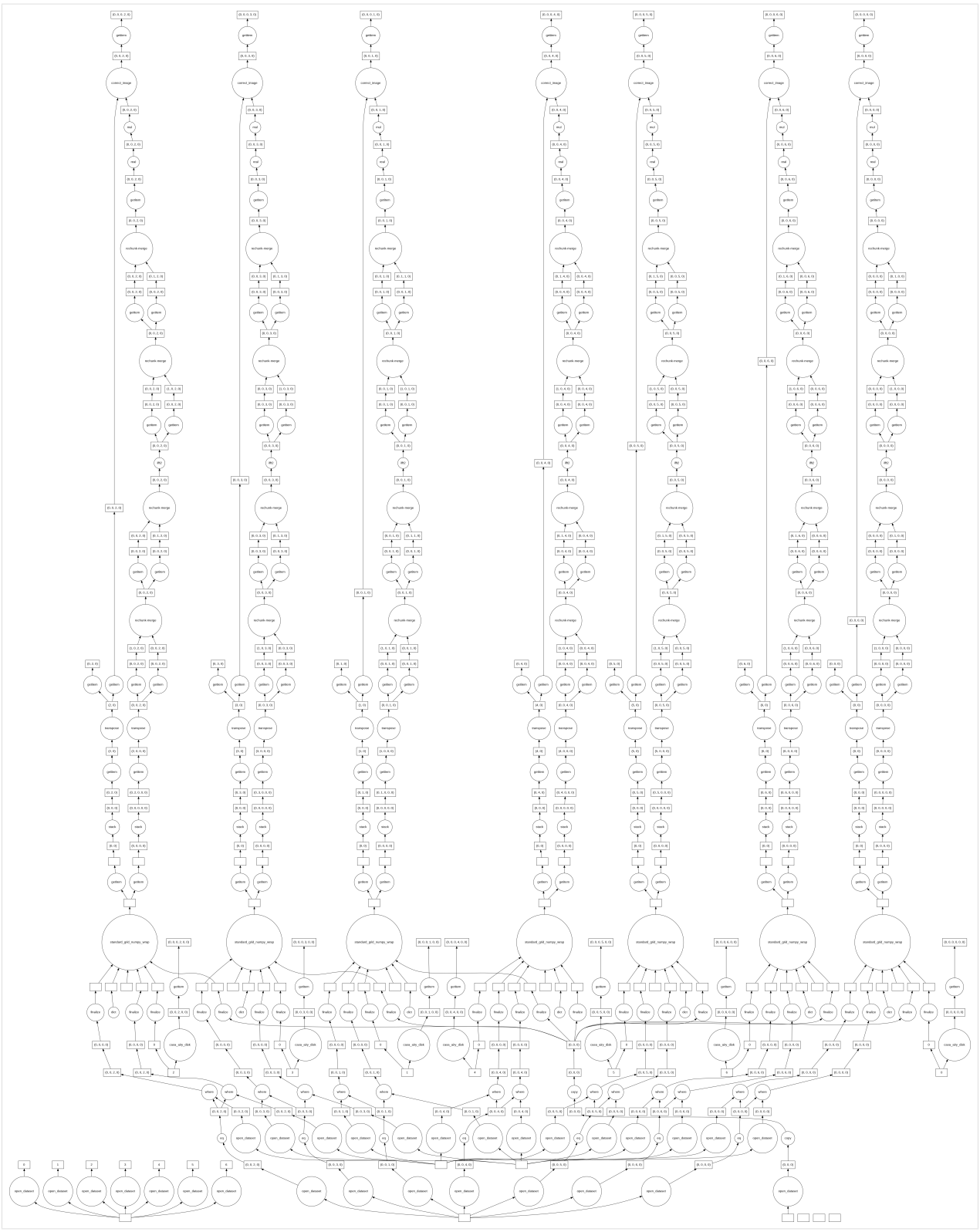
```
######################### Start make_image #########################
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '1', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0'}
Setting default data_group_out [' sum_weight ']  to  SUM_WEIGHT
Setting default data_group_out [' image ']  to  IMAGE
######################### Created graph for make_image #########################
######################### Start make_pb #########################
```

(continues on next page)

(continued from previous page)

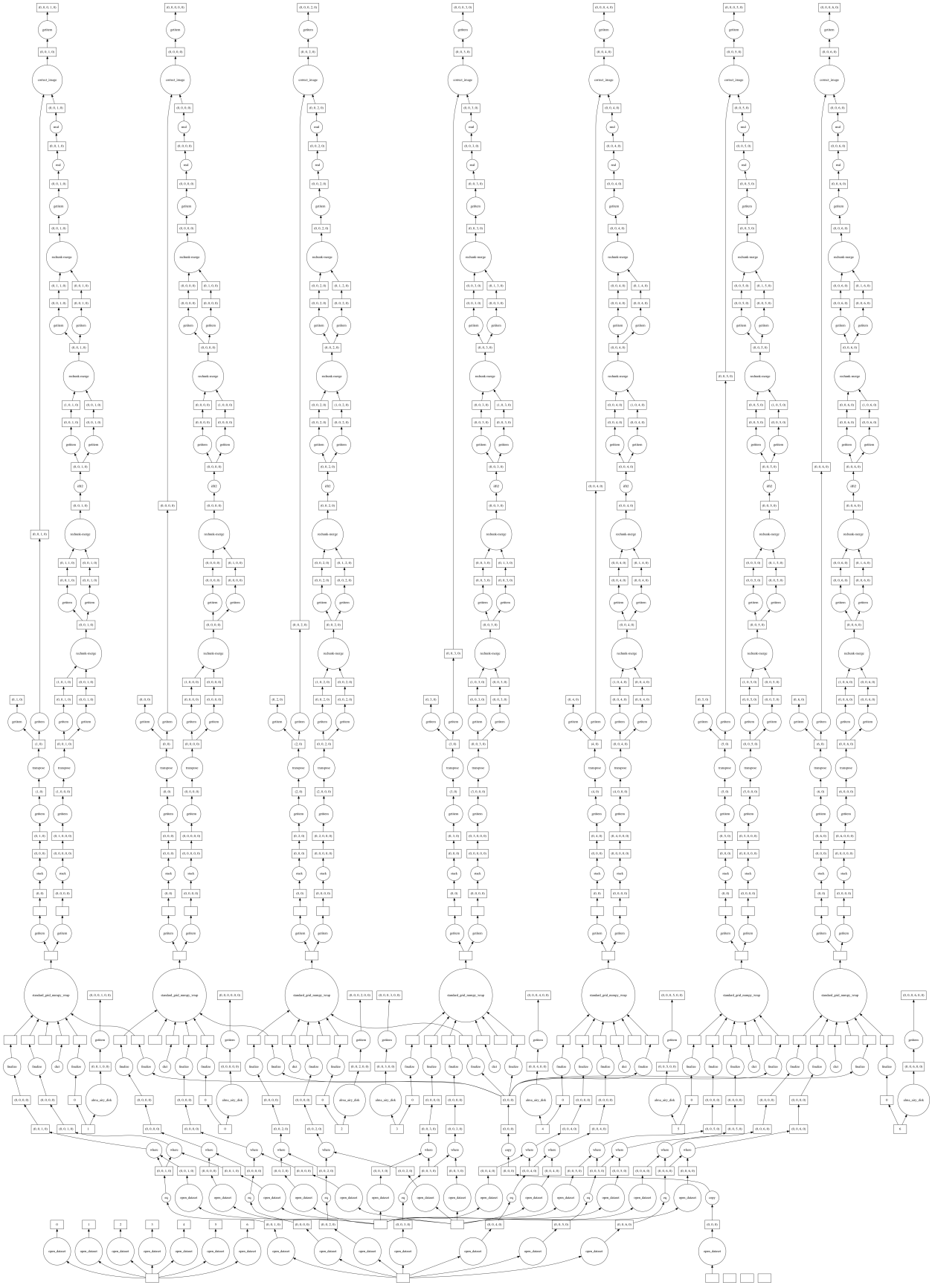
```
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT', 'image':
↪'IMAGE'}
Setting default data_group_out  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT', 'image':
↪'IMAGE', 'pb': 'PB'}
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
######################### Created graph for make_pb ########################
```

[5]:



8.2.5 Dask Visualization

The Dask execution graph below shows how the images for each channel are computed in parallel. Each image is written to disk independently and Dask along with Zarr handles the virtual concatenation (the resulting img.zarr is chunked by channel). This allows for processing cubes that are larger than memory.



8.2.6 Save Image to Disk (execute graph)

[6]:
```
img_xds = write_image(img_xds, outfile='twhya_standard_gridder_lsrk_cube_natural.img.
↪zarr', graph_name='make_imaging_weights, make_image and make_pb')
```

```
Time to store and execute graph  make_imaging_weights, make_image and make_pb 3.
↪455151081085205
```

8.2.7 Plot and Compare With CASA

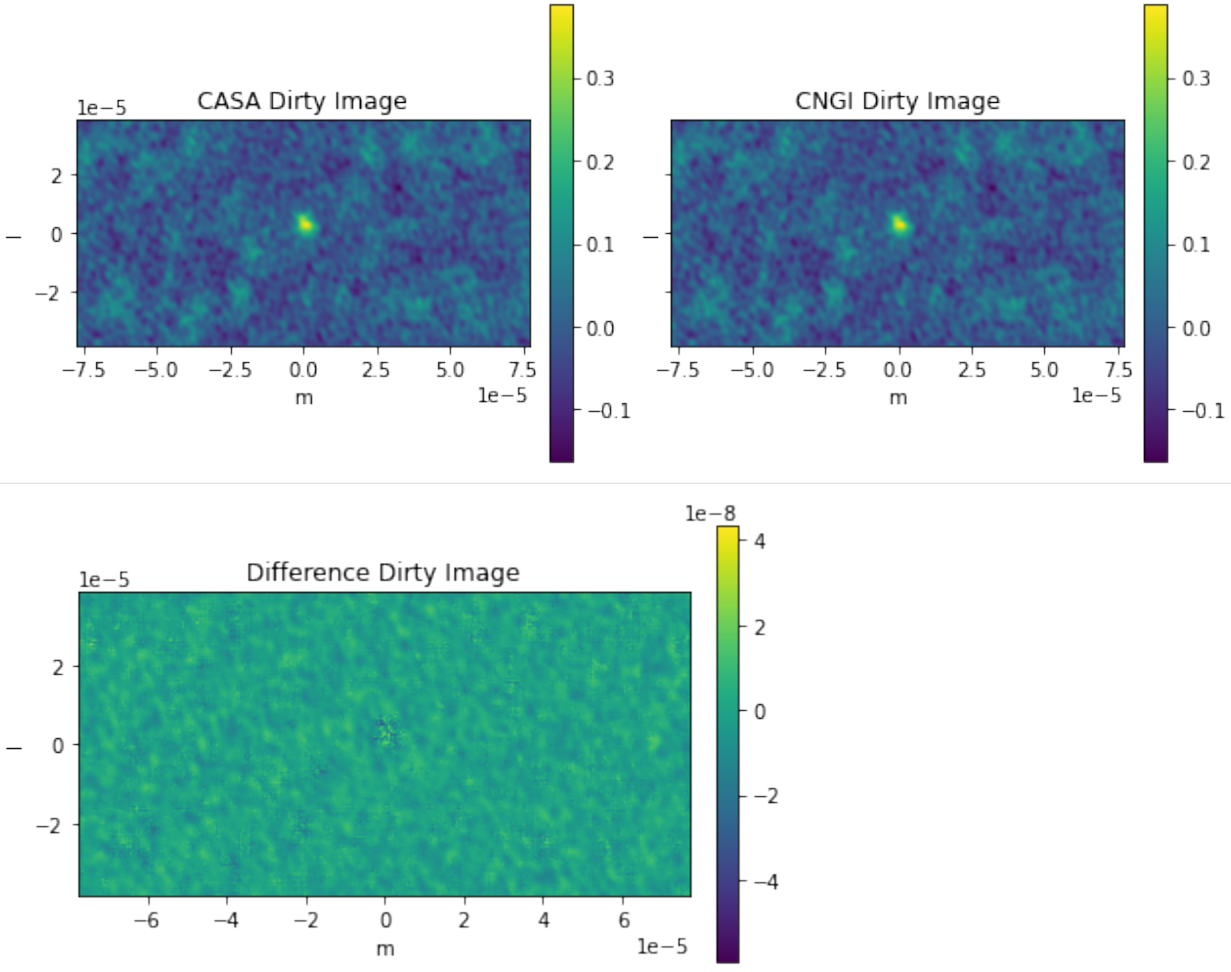
[7]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive
import xarray as xr


def comparison_plots_1(chan):
    img_xds = xr.open_zarr('twhya_standard_gridder_lsrk_cube_natural.img.zarr').
↪isel(time=0,pol=0)
    casa_img_xds = read_image("casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr
↪").isel(time=0,pol=0)
    plt.close('all')
    print('Frequency',img_xds.chan[chan].values, 'Hz')
    #print(img_xds['IMAGE'])
    dirty_image = img_xds['IMAGE'].isel(chan=chan)
    casa_dirty_image = casa_img_xds['RESIDUAL'].isel(chan=chan)

    minmin = np.min([np.min(casa_dirty_image.data), np.min(dirty_image.data)])
    maxmax = np.max([np.max(casa_dirty_image.data), np.max(dirty_image.data)])
    extent = extent=(np.min(casa_dirty_image.m),np.max(casa_dirty_image.m),np.
↪min(casa_dirty_image.l),np.max(casa_dirty_image.l))

    fig0, ax0 = plt.subplots(1, 2, sharey=True, figsize=(10, 5))
    im0 = ax0[0].imshow(casa_dirty_image,vmin=minmin,vmax=maxmax,extent=extent)
    im1 = ax0[1].imshow(dirty_image,vmin=minmin, vmax=maxmax,extent=extent)
    ax0[0].title.set_text('CASA Dirty Image')
    ax0[1].title.set_text('CNGI Dirty Image')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'),ax0[1].set_
↪ylabel('l')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
    plt.show()

    plt.figure()
    plt.imshow(casa_dirty_image - dirty_image,extent=extent)
    plt.title('Difference Dirty Image')
    plt.xlabel('m')
    plt.ylabel('l')
    plt.colorbar(fraction=0.046, pad=0.04)
    plt.show()

    dirty_image = dirty_image / np.max(np.abs(dirty_image))
    casa_dirty_image = casa_dirty_image / np.max(np.abs(casa_dirty_image))

    # Calculate max error
    max_error_dirty_image = np.max(np.abs(dirty_image - casa_dirty_image)).values
```

(continues on next page)

(continued from previous page)

```
    print('Max Error',max_error_dirty_image)
    # Calculate root mean square error
    rms_error_dirty_image = np.linalg.norm(dirty_image - casa_dirty_image, 'fro')
    print('RMS Error',rms_error_dirty_image)



#interactive_plot_1 = interactive(comparison_plots_1, chan=(0, 6))
#output_1 = interactive_plot_1.children[-1]
#output_1.layout.auto_scroll_threshold = 9999;
#interactive_plot_1

comparison_plots_1(3)
```

```
Frequency 372521853594.1145 Hz
```




```
Max Error 1.662581979866573e-07
RMS Error 4.023737036016809e-06
```

The first channel (channel 0) is flagged by both ngCASA and CASA. Why CASA is flagging the last channel and ngCASA is not, this needs further investigation. Checking sis14_twhya_chan_avg_field_5_lsrk_pol_xx.ms with browsetable in CASA shows that only the first channel is flagged.

The reason for the small difference between ngCASA and CASA, in channels 1 to 5, is due to ngCASA using a

different implementation of the Fast Fourier Transform.

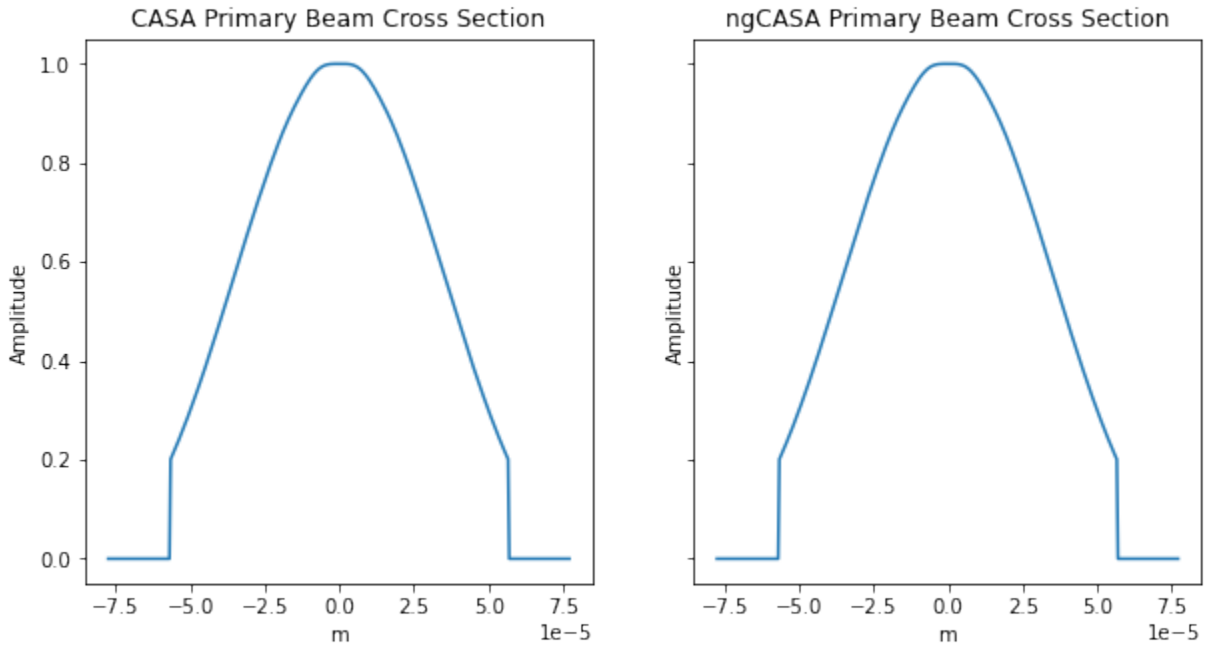
[8]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive


#### Primary Beam Corrected Images ####
def comparison_plots_2(chan):
    img_xds = xr.open_zarr('twhya_standard_gridder_lsrk_cube_natural.img.zarr').
↪isel(time=0,pol=0, dish_type=0)
    casa_img_xds = read_image("casa_twhya_standard_gridder_lsrk_cube_natural.img.zarr
↪").isel(time=0, pol=0)
    extent = extent=(np.min(casa_img_xds.m),np.max(casa_img_xds.m),np.min(casa_img_
↪xds.l),np.max(casa_img_xds.l))

    plt.close('all')
    print('Frequency',img_xds.chan[chan].values, 'Hz')
    pb_limit = 0.2
    primary_beam = img_xds.PB.isel(l=100,chan=chan).where(img_xds.PB.isel(l=100,
↪chan=chan) > pb_limit,other=0.0)
    dirty_image_pb_cor = img_xds.IMAGE.isel(chan=chan)/img_xds.PB.isel(chan=chan)
    dirty_image_pb_cor = dirty_image_pb_cor.where(img_xds.PB.isel(chan=chan) > pb_
↪limit,other=np.nan)

    casa_primary_beam = casa_img_xds['PB'].isel(l=100,chan=chan) #Primary beam␣
↪created by CASA
    casa_dirty_image_pb_cor = (casa_img_xds['IMAGE_PBCOR'].isel(chan=chan)).
↪where(casa_img_xds['PB'].isel(chan=chan) > pb_limit,other=np.nan) #Image created by␣
↪CASA


    #Plot Primary Beams
    fig0, ax0, = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
    im0 = ax0[0].plot(casa_img_xds.m,casa_primary_beam)
    im1 = ax0[1].plot(casa_img_xds.m,primary_beam)
    ax0[0].title.set_text('CASA Primary Beam Cross Section')
    ax0[1].title.set_text('ngCASA Primary Beam Cross Section')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m')
    ax0[0].set_ylabel('Amplitude'), ax0[1].set_ylabel('Amplitude')
    plt.show()

    plt.figure()
    plt.plot(casa_img_xds.m,casa_primary_beam-primary_beam)
    plt.title('Difference Primary Beam')
    plt.xlabel('m')
    plt.ylabel('Amplitude')
    plt.show()

    #Plotting Images
    fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
    im0 = ax0[0].imshow(casa_dirty_image_pb_cor,extent=extent)
    im1 = ax0[1].imshow(dirty_image_pb_cor,extent=extent)
    ax0[0].title.set_text('CASA PB Corrected Dirty Image')
    ax0[1].title.set_text('ngCASA PB Corrected Dirty Image')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'), ax0[1].
↪set_ylabel('l')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
```

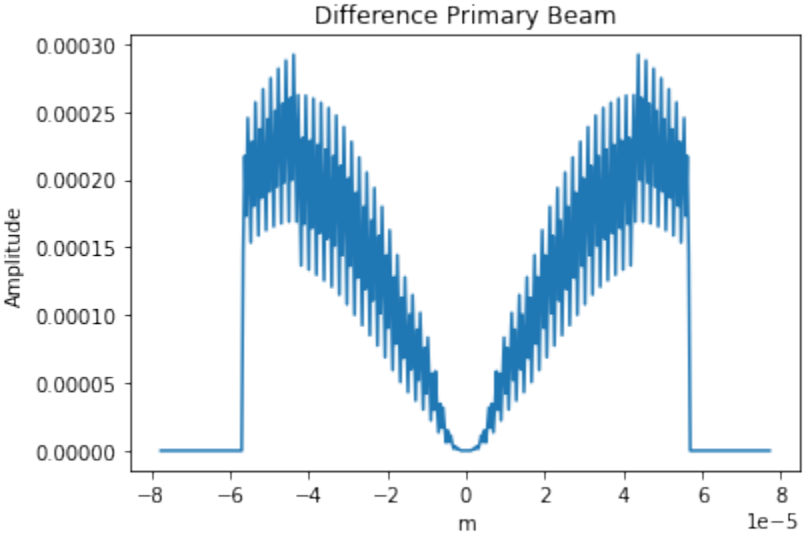
(continues on next page)

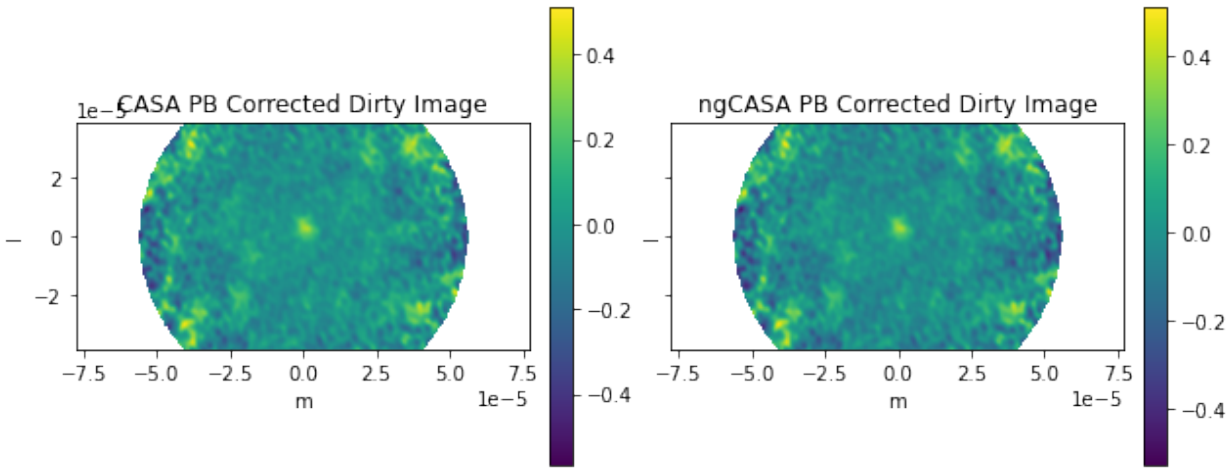
(continued from previous page)

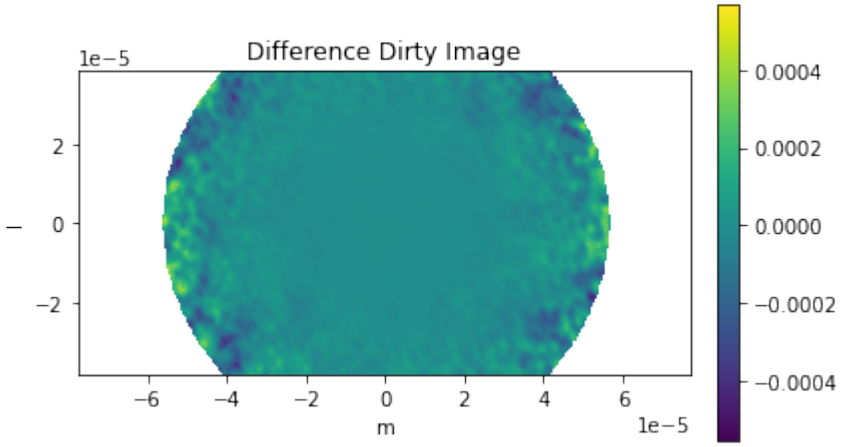
```
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
    plt.show()

    plt.figure()
    plt.imshow(casa_dirty_image_pb_cor - dirty_image_pb_cor,extent=extent)
    plt.title('Difference Dirty Image')
    plt.xlabel('m'), plt.ylabel('l')
    plt.colorbar(fraction=0.046, pad=0.04)
    plt.show()

    dirty_image_pb_cor = dirty_image_pb_cor / np.nanmax(np.abs(dirty_image_pb_cor))
    casa_dirty_image_pb_cor = casa_dirty_image_pb_cor / np.nanmax(np.abs(casa_dirty_
↪image_pb_cor))
    norm_diff_image_pb_cor = dirty_image_pb_cor - casa_dirty_image_pb_cor

    # Calculate max error
    max_error_dirty_image = np.nanmax(np.abs(norm_diff_image_pb_cor))
    print('Max Normalized Error',max_error_dirty_image)
    # Calculate root mean square error
    rms_error_dirty_image = np.sqrt(np.nansum(np.square(norm_diff_image_pb_cor)))
    print('RMS Normalized Error',rms_error_dirty_image)


#interactive_plot_2 = interactive(comparison_plots_2, chan=(0, 6))
#output_2 = interactive_plot_2.children[-1]
#output_2.layout.auto_scroll_threshold = 9999;
#interactive_plot_2

comparison_plots_2(3)
```

```
Frequency 372521853594.1145 Hz
```










```
Max Normalized Error 0.06926086996721081
RMS Normalized Error 3.0796672817323736
```

The frequency does not change enough for the primary beam to vary significantly.

The difference in primary beam is due to CASA using a sampled 1D function while ngCASA calculates the PB for each pixel. If it is found that PB creation becomes a bottleneck for ngCASA the implementation will be changed to match CASA.

8.2.8 synthesis_imaging_cube function

The synthesis_imaging_cube is an standard gridder imager that combines the flagging and the creation of the imagining weights, primary beam, PSF, dirty image into a single function. The advantage of this function is that it simplifies the graph by combining functions that can reside in the same node.

[9]:
```
%load_ext autoreload
%autoreload 2
from ngcasa.imaging import synthesis_imaging_cube
from cngi.dio import read_vis, write_image
import xarray as xr

mxds = read_vis("sis14_twhya_chan_avg_field_5_lsrk_pol_xx.vis.zarr")
print(mxds.xds0)

grid_parms = {}
grid_parms['chan_mode'] = 'cube'
grid_parms['image_size'] = [200,400]
grid_parms['cell_size'] = [0.08,0.08]
grid_parms['phase_center'] = mxds.FIELD.PHASE_DIR[0,0,:].data.compute()

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'
#imaging_weights_parms['weighting'] = 'briggs'
#imaging_weights_parms['robust'] = 0.5

vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 0

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 0

make_pb_parms = {}
make_pb_parms['function'] = 'casa_airy'
make_pb_parms['list_dish_diameters'] = [10.7]
make_pb_parms['list_blockage_diameters'] = [0.75]


img_xds2 = xr.Dataset()
img_xds2 = synthesis_imaging_cube(mxds, img_xds2, grid_parms, imaging_weights_parms,␣
↪make_pb_parms, vis_sel_parms, img_sel_parms)
write_image(img_xds2, outfile='twhya_standard_gridder_lsrk_cube_natural2.img.zarr',␣
↪graph_name='synthesis_imaging_cube')
```

```
overwrite_encoded_chunks True
<xarray.Dataset>
Dimensions:          (baseline: 210, chan: 7, pol: 1, pol_id: 1, spw_id: 1, time: 270,␣
↪uvw_index: 3)
Coordinates:
  * baseline         (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
  * chan             (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11 3.725e+11
    chan_width       (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    effective_bw    (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * pol             (pol) int32 9
  * pol_id          (pol_id) int32 0
    resolution      (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * spw_id          (spw_id) int32 0
  * time            (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1        (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2        (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID        (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    DATA            (time, baseline, chan, pol) complex128 dask.array<chunksize=(270,␣
↪210, 1, 1), meta=np.ndarray>
    DATA_WEIGHT     (time, baseline, chan, pol) float64 dask.array<chunksize=(270,␣
↪210, 1, 1), meta=np.ndarray>
    EXPOSURE        (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    ...              ...
    OBSERVATION_ID  (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    PROCESSOR_ID    (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    SCAN_NUMBER     (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    STATE_ID        (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    TIME_CENTROID   (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    UVW             (time, baseline, uvw_index) float64 dask.array<chunksize=(270,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         7
    num_corr:         1
    ref_frequency:    372520022603.63745
    total_bandwidth:  4272311.112915039
v3
######################### Start Synthesis Imaging Cube ########################
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '1', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0'}
Setting default data_group_out [' image_sum_weight ']  to  IMAGE_SUM_WEIGHT
Setting default data_group_out [' image ']  to  IMAGE
Setting default data_group_out [' psf_sum_weight ']  to  PSF_SUM_WEIGHT
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_out [' psf ']  to  PSF
Setting default data_group_out [' pb ']  to  PB
Setting default data_group_out [' restore_parms ']  to  RESTORE_PARMS
dask.array<concatenate, shape=(7, 1, 3), dtype=float64, chunksize=(1, 1, 3),␣
↪chunktype=numpy.ndarray>
Time to store and execute graph  synthesis_imaging_cube 11.652380228042603
```

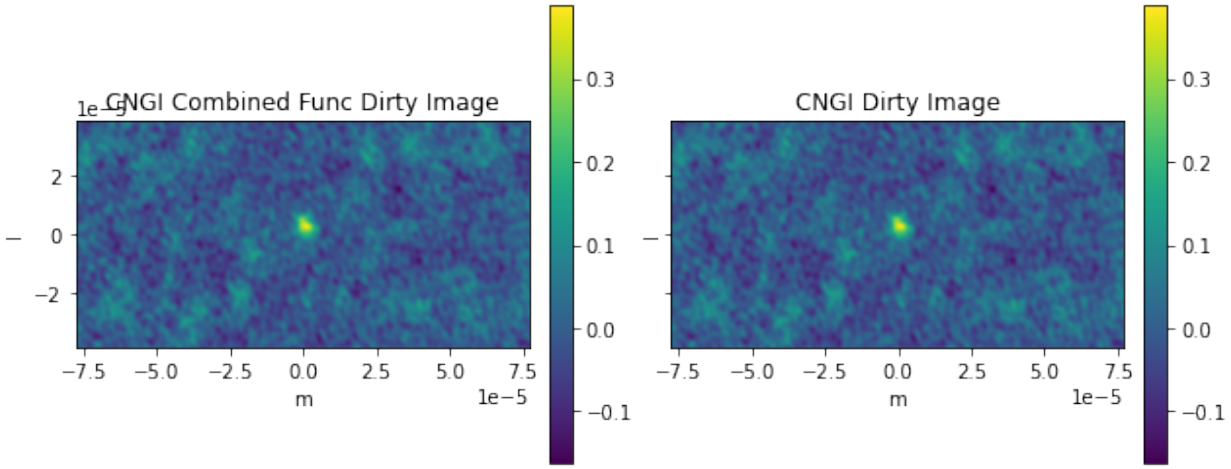
[10]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive
import xarray as xr


def comparison_plots_1(chan):
    img_xds = xr.open_zarr('twhya_standard_gridder_lsrk_cube_natural.img.zarr').
↪isel(time=0,pol=0)
    img_xds2 = read_image("twhya_standard_gridder_lsrk_cube_natural2.img.zarr").
↪isel(time=0,pol=0)
    plt.close('all')
    print('Frequency',img_xds.chan[chan].values, 'Hz')
    #print(img_xds['IMAGE'])
    dirty_image = img_xds['IMAGE'].isel(chan=chan)
    dirty_image2 = img_xds2['IMAGE'].isel(chan=chan)

    minmin = np.min([np.min(dirty_image2.data), np.min(dirty_image.data)])
    maxmax = np.max([np.max(dirty_image2.data), np.max(dirty_image.data)])
    extent = extent=(np.min(dirty_image2.m),np.max(dirty_image2.m),np.min(dirty_
↪image2.l),np.max(dirty_image2.l))

    fig0, ax0 = plt.subplots(1, 2, sharey=True, figsize=(10, 5))
    im0 = ax0[0].imshow(dirty_image2,vmin=minmin,vmax=maxmax,extent=extent)
    im1 = ax0[1].imshow(dirty_image,vmin=minmin, vmax=maxmax,extent=extent)
    ax0[0].title.set_text('CNGI Combined Func Dirty Image')
    ax0[1].title.set_text('CNGI Dirty Image')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'),ax0[1].set_
↪ylabel('l')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
    plt.show()

    plt.figure()
    plt.imshow(dirty_image2 - dirty_image,extent=extent)
    plt.title('Difference Dirty Image')
    plt.xlabel('m')
    plt.ylabel('l')
    plt.colorbar(fraction=0.046, pad=0.04)
    plt.show()

    dirty_image = dirty_image / np.max(np.abs(dirty_image))
    dirty_image2 = dirty_image2 / np.max(np.abs(dirty_image2))

    # Calculate max error
    max_error_dirty_image = np.max(np.abs(dirty_image - dirty_image2)).values
    print('Max Error',max_error_dirty_image)
    # Calculate root mean square error
    rms_error_dirty_image = np.linalg.norm(dirty_image - dirty_image2, 'fro')
```

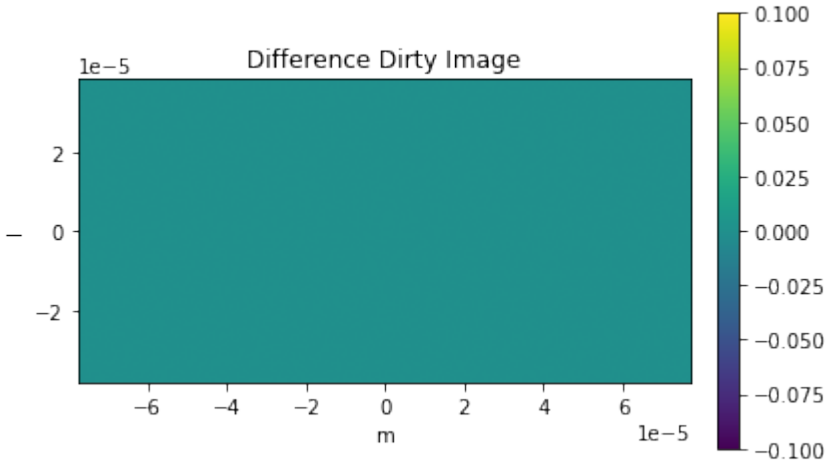
(continues on next page)

(continued from previous page)

```
    print('RMS Error',rms_error_dirty_image)



#interactive_plot_1 = interactive(comparison_plots_1, chan=(0, 6))
#output_1 = interactive_plot_1.children[-1]
#output_1.layout.auto_scroll_threshold = 9999;
#interactive_plot_1

comparison_plots_1(3)
```

```
Frequency 372521853594.1145 Hz
```






```
Max Error 0.0
RMS Error 0.0
```

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/imaging/imaging_weights_example.ipynb

8.3 Imaging Weights

This notebook will demonstrate how to create images with different imaging weighting schemes (natural, uniform, briggs) and how to make use of the `storage_parms` to append images.

For this demonstration data from the ALMA First Look at Imaging CASAguide (https://casaguides.nrao.edu/index.php/First_Look_at_Imaging) will be used. The measurement set has been converted to vis.zarr (using convert_ms in cngi.conversion)

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here.

8.3.1 Installation and Dataset Download

[1]:
```
import os
os.system("pip install cngi-prototype==0.0.91")

!gdown -q --id 1PNL0ANqnN7eyYOpQ_vMslQlorUhtrUmE
!unzip sis14_twhya_field_5_lsrk_pol_xx.vis.zarr.zip > /dev/null

%matplotlib widget
print('complete')
```

```
complete
```

8.3.2 Load Dataset

For an explanation of how the vis.zarr file was created and the chunking go to the continuum image example.

[2]:
```
import xarray as xr
from cngi.dio import read_vis

xr.set_options(display_style="html")

mxds = read_vis("sis14_twhya_field_5_lsrk_pol_xx.vis.zarr",chunks={'chan':192})
print(mxds.xds0)
```

```
overwrite_encoded_chunks True
<xarray.Dataset>
Dimensions:          (baseline: 210, chan: 384, pol: 1, pol_id: 1, spw_id: 1, time:␣
↪270, uvw_index: 3)
Coordinates:
  * baseline         (baseline) int64 0 1 2 3 4 5 6 ... 204 205 206 207 208 209
  * chan             (chan) float64 3.725e+11 3.725e+11 ... 3.728e+11 3.728e+11
    chan_width       (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
    effective_bw     (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * pol              (pol) int32 9
  * pol_id           (pol_id) int32 0
    resolution       (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * spw_id           (spw_id) int32 0
  * time             (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1         (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2         (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    ARRAY_ID         (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    DATA             (time, baseline, chan, pol) complex128 dask.array<chunksize=(270,␣
↪210, 192, 1), meta=np.ndarray>
    DATA_WEIGHT      (time, baseline, chan, pol) float64 dask.array<chunksize=(270,␣
↪210, 192, 1), meta=np.ndarray>
    EXPOSURE         (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    ...               ...
    OBSERVATION_ID   (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    PROCESSOR_ID     (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    SCAN_NUMBER      (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    STATE_ID         (time, baseline) int32 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    TIME_CENTROID    (time, baseline) float64 dask.array<chunksize=(270, 210), meta=np.
↪ndarray>
    UVW              (time, baseline, uvw_index) float64 dask.array<chunksize=(270,␣
↪210, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         384
    num_corr:         1
    ref_frequency:    372520022603.63745
    total_bandwidth:  234366781.0546875
```

8.3.3 Make Imaging Weights

make_imaging_weight documentation

[3]:
```
from ngcasa.imaging import make_imaging_weight
from cngi.vis import apply_flags
from ngcasa.imaging import make_psf
from ngcasa.imaging import make_image

grid_parms = {}
grid_parms['chan_mode'] = 'continuum'
grid_parms['image_size'] = [200,400]
grid_parms['cell_size'] = [0.08,0.08]

mxds_0 = apply_flags(mxds, 'xds0', flags='FLAG')

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'
```

(continues on next page)

(continued from previous page)

```
sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0
sel_parms['data_group_out_id'] = 0

mxds_1 = make_imaging_weight(mxds_0, imaging_weights_parms, grid_parms, sel_parms)

##################
imaging_weights_parms['weighting'] = 'uniform'

sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0
sel_parms['data_group_out_id'] = 1
sel_parms['imaging_weight'] = 'IMAGING_WEIGHT_UNI'

mxds_2 = make_imaging_weight(mxds_1, imaging_weights_parms, grid_parms, sel_parms)

##################
imaging_weights_parms['weighting'] = 'briggs'
imaging_weights_parms['robust'] = 0.6 #number, default:0.5, acceptable values [-2,2]

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0
sel_parms['data_group_out_id'] = 2
sel_parms['imaging_weight'] = 'IMAGING_WEIGHT_BRG'

mxds_3 = make_imaging_weight(mxds_2, imaging_weights_parms, grid_parms, sel_parms)

print(mxds_3.xds0)
```

```
######################### Start make_imaging_weights #########################
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out [' data ']  to  DATA
Setting default data_group_out [' flag ']  to  FLAG
Setting default data_group_out [' uvw ']  to  UVW
Setting default data_group_out [' weight ']  to  DATA_WEIGHT
Setting default data_group_out [' imaging_weight ']  to  IMAGING_WEIGHT
Since weighting is natural input weight will be reused as imaging weight.
######################### Created graph for make_imaging_weight ######################
↪###
######################### Start make_imaging_weights #########################
Setting data_group_in  to  {'id': '0', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out [' data ']  to  DATA
Setting default data_group_out [' flag ']  to  FLAG
Setting default data_group_out [' uvw ']  to  UVW
Setting default data_group_out [' weight ']  to  DATA_WEIGHT
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
######################### Created graph for make_imaging_weight ######################
↪###
######################### Start make_imaging_weights #########################
Setting data_group_in  to  {'id': '0', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out [' data ']  to  DATA
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_out [' flag ']  to  FLAG
Setting default data_group_out [' uvw ']  to  UVW
Setting default data_group_out [' weight ']  to  DATA_WEIGHT
Setting default image_center  to  [100 200]
Setting default fft_padding  to  1.2
######################### Created graph for make_imaging_weight ####################
↪###
<xarray.Dataset>
Dimensions:              (baseline: 210, chan: 384, pol: 1, pol_id: 1, spw_id: 1, time:
↪ 270, uvw_index: 3)
Coordinates:
  * baseline             (baseline) int64 0 1 2 3 4 5 ... 204 205 206 207 208 209
  * chan                 (chan) float64 3.725e+11 3.725e+11 ... 3.728e+11
  * pol                  (pol) int32 9
  * time                 (time) datetime64[ns] 2012-11-19T07:56:26.544000626 ...
    chan_width           (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
    effective_bw         (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * pol_id               (pol_id) int32 0
    resolution           (chan) float64 dask.array<chunksize=(192,), meta=np.ndarray>
  * spw_id               (spw_id) int32 0
Dimensions without coordinates: uvw_index
Data variables: (12/19)
    ANTENNA1             (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ANTENNA2             (baseline) int32 dask.array<chunksize=(210,), meta=np.ndarray>
    ARRAY_ID             (time, baseline) int32 dask.array<chunksize=(270, 210),␣
↪meta=np.ndarray>
    DATA                 (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(270, 210, 192, 1), meta=np.ndarray>
    DATA_WEIGHT          (time, baseline, chan, pol) float64 dask.array<chunksize=(270,
↪ 210, 192, 1), meta=np.ndarray>
    EXPOSURE             (time, baseline) float64 dask.array<chunksize=(270, 210),␣
↪meta=np.ndarray>
    ...                   ...
    SCAN_NUMBER          (time, baseline) int32 dask.array<chunksize=(270, 210),␣
↪meta=np.ndarray>
    STATE_ID             (time, baseline) int32 dask.array<chunksize=(270, 210),␣
↪meta=np.ndarray>
    TIME_CENTROID        (time, baseline) float64 dask.array<chunksize=(270, 210),␣
↪meta=np.ndarray>
    UVW                  (time, baseline, uvw_index) float64 dask.array<chunksize=(270,
↪ 210, 3), meta=np.ndarray>
    IMAGING_WEIGHT_UNI   (time, baseline, chan, pol) float64 dask.array<chunksize=(270,
↪ 210, 192, 1), meta=np.ndarray>
    IMAGING_WEIGHT_BRG   (time, baseline, chan, pol) float64 dask.array<chunksize=(270,
↪ 210, 192, 1), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           2
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'id': '0', 'data': 'DATA', 'flag': 'FLAG', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:             ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:     2
    num_chan:         384
    num_corr:         1
```

(continues on next page)

(continued from previous page)

```
    ref_frequency:    372520022603.63745
    total_bandwidth:  234366781.0546875
```

8.3.4 Make Image and PSF

make_psf documentation

[4]:
```
from ngcasa.imaging import make_image
from ngcasa.imaging import make_pb
from cngi.dio import write_image
import dask
import xarray as xr

grid_parms = {}
grid_parms['chan_mode'] = 'continuum'
grid_parms['image_size'] = [200,200]
grid_parms['cell_size'] = [0.04,0.04]
grid_parms['phase_center'] = mxds.FIELD.PHASE_DIR[0,0,:].data.compute()


############################## Natural
vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 0

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 0
img_sel_parms['psf'] = 'PSF_NAT'
img_sel_parms['psf_sum_weight'] = 'PSF_SUM_WEIGHT_NAT'
img_sel_parms['image'] = 'IMAGE_NAT'
img_sel_parms['sum_weight'] = 'SUM_WEIGHT_NAT'
img_sel_parms['psf_fit'] = 'PSF_FIT_NAT'

img_xds = xr.Dataset() #empty dataset
img_xds = make_psf(mxds_3, img_xds, grid_parms, vis_sel_parms, img_sel_parms)
img_xds = make_image(mxds_3, img_xds, grid_parms, vis_sel_parms, img_sel_parms)

############################## Uniform
vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 1

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 1
img_sel_parms['psf'] = 'PSF_UNI'
img_sel_parms['psf_sum_weight'] = 'PSF_SUM_WEIGHT_UNI'
img_sel_parms['image'] = 'IMAGE_UNI'
img_sel_parms['sum_weight'] = 'SUM_WEIGHT_UNI'
img_sel_parms['psf_fit'] = 'PSF_FIT_UNI'

img_xds = make_psf(mxds_3, img_xds, grid_parms, vis_sel_parms, img_sel_parms)
img_xds = make_image(mxds_3, img_xds, grid_parms, vis_sel_parms, img_sel_parms)

############################## Briggs
vis_sel_parms = {}
```

(continues on next page)

(continued from previous page)

```
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 2

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 2
img_sel_parms['psf'] = 'PSF_BRG'
img_sel_parms['psf_sum_weight'] = 'PSF_SUM_WEIGHT_BRG'
img_sel_parms['image'] = 'IMAGE_BRG'
img_sel_parms['sum_weight'] = 'SUM_WEIGHT_BRG'
img_sel_parms['psf_fit'] = 'PSF_FIT_BRG'

img_xds = make_psf(mxds_3, img_xds, grid_parms, vis_sel_parms, img_sel_parms)
img_xds = make_image(mxds_3, img_xds, grid_parms, vis_sel_parms, img_sel_parms)

print(img_xds)
```

```
######################### Start make_psf #########################
Setting default image_center  to  [100 100]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'id': '0', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'id': '3', 'data': 'DATA', 'flag': 'FLAG', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0'}
######################### Created graph for make_psf #########################
######################### Start make_image #########################
Setting default image_center  to  [100 100]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'id': '0', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'id': '3', 'data': 'DATA', 'flag': 'FLAG', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT_NAT', 'psf':
↪'PSF_NAT', 'psf_fit': 'PSF_FIT_NAT'}
Setting default data_group_out [' psf ']  to  PSF_NAT
Setting default data_group_out [' psf_fit ']  to  PSF_FIT_NAT
######################### Created graph for make_image #########################
######################### Start make_psf #########################
Setting default image_center  to  [100 100]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'id': '1', 'imaging_weight': 'IMAGING_WEIGHT_UNI', 'data':
↪ 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'id': '3', 'imaging_weight': 'IMAGING_WEIGHT_UNI
↪', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT_NAT', 'image
↪': 'IMAGE_NAT', 'psf': 'PSF_NAT', 'psf_fit': 'PSF_FIT_NAT'}
Setting default data_group_out [' image ']  to  IMAGE_NAT
######################### Created graph for make_psf #########################
######################### Start make_image #########################
Setting default image_center  to  [100 100]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'id': '1', 'imaging_weight': 'IMAGING_WEIGHT_UNI', 'data':
↪ 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'id': '3', 'imaging_weight': 'IMAGING_WEIGHT_UNI
↪', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT_NAT', 'image
↪': 'IMAGE_NAT', 'psf': 'PSF_NAT', 'psf_fit': 'PSF_FIT_NAT'}
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_out [' psf ']  to  PSF_NAT
Setting default data_group_out [' psf_fit ']  to  PSF_FIT_NAT
######################### Created graph for make_image ########################
######################### Start make_psf ########################
Setting default image_center  to  [100 100]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'id': '2', 'imaging_weight': 'IMAGING_WEIGHT_BRG', 'data':
↪ 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'id': '3', 'imaging_weight': 'IMAGING_WEIGHT_BRG
↪', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT_NAT', 'image
↪': 'IMAGE_NAT', 'psf': 'PSF_NAT', 'psf_fit': 'PSF_FIT_NAT'}
Setting default data_group_out [' image ']  to  IMAGE_NAT
######################### Created graph for make_psf ########################
######################### Start make_image ########################
Setting default image_center  to  [100 100]
Setting default fft_padding  to  1.2
Setting data_group_in  to  {'id': '2', 'imaging_weight': 'IMAGING_WEIGHT_BRG', 'data':
↪ 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'id': '3', 'imaging_weight': 'IMAGING_WEIGHT_BRG
↪', 'data': 'DATA', 'flag': 'FLAG', 'uvw': 'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_in  to  {'id': '0', 'sum_weight': 'SUM_WEIGHT_NAT', 'image
↪': 'IMAGE_NAT', 'psf': 'PSF_NAT', 'psf_fit': 'PSF_FIT_NAT'}
Setting default data_group_out [' psf ']  to  PSF_NAT
Setting default data_group_out [' psf_fit ']  to  PSF_FIT_NAT
######################### Created graph for make_image ########################
<xarray.Dataset>
Dimensions:           (chan: 1, elps_index: 3, l: 200, m: 200, pol: 1, time: 1)
Coordinates:
  * chan              (chan) float64 3.726e+11
  * l                 (l) float64 1.939e-05 1.92e-05 ... -1.9e-05 -1.92e-05
  * m                 (m) float64 -1.939e-05 -1.92e-05 ... 1.9e-05 1.92e-05
  * pol               (pol) int32 9
  * time              (time) datetime64[ns] 2012-11-19T08:26:09.618666673
    chan_width        (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    right_ascension   (l, m) float64 2.888 2.888 2.888 ... 2.888 2.888 2.888
    declination       (l, m) float64 -0.6057 -0.6057 -0.6057 ... -0.6057 -0.6057
Dimensions without coordinates: elps_index
Data variables:
    SUM_WEIGHT_NAT    (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    PSF_NAT           (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 200,␣
↪1, 1, 1), meta=np.ndarray>
    PSF_FIT_NAT       (time, chan, pol, elps_index) float64 dask.array<chunksize=(1, 1,
↪ 1, 3), meta=np.ndarray>
    IMAGE_NAT         (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 200,␣
↪1, 1, 1), meta=np.ndarray>
    SUM_WEIGHT_UNI    (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    PSF_UNI           (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 200,␣
↪1, 1, 1), meta=np.ndarray>
    PSF_FIT_UNI       (time, chan, pol, elps_index) float64 dask.array<chunksize=(1, 1,
↪ 1, 3), meta=np.ndarray>
    IMAGE_UNI         (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 200,␣
↪1, 1, 1), meta=np.ndarray>
    SUM_WEIGHT_BRG    (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
```

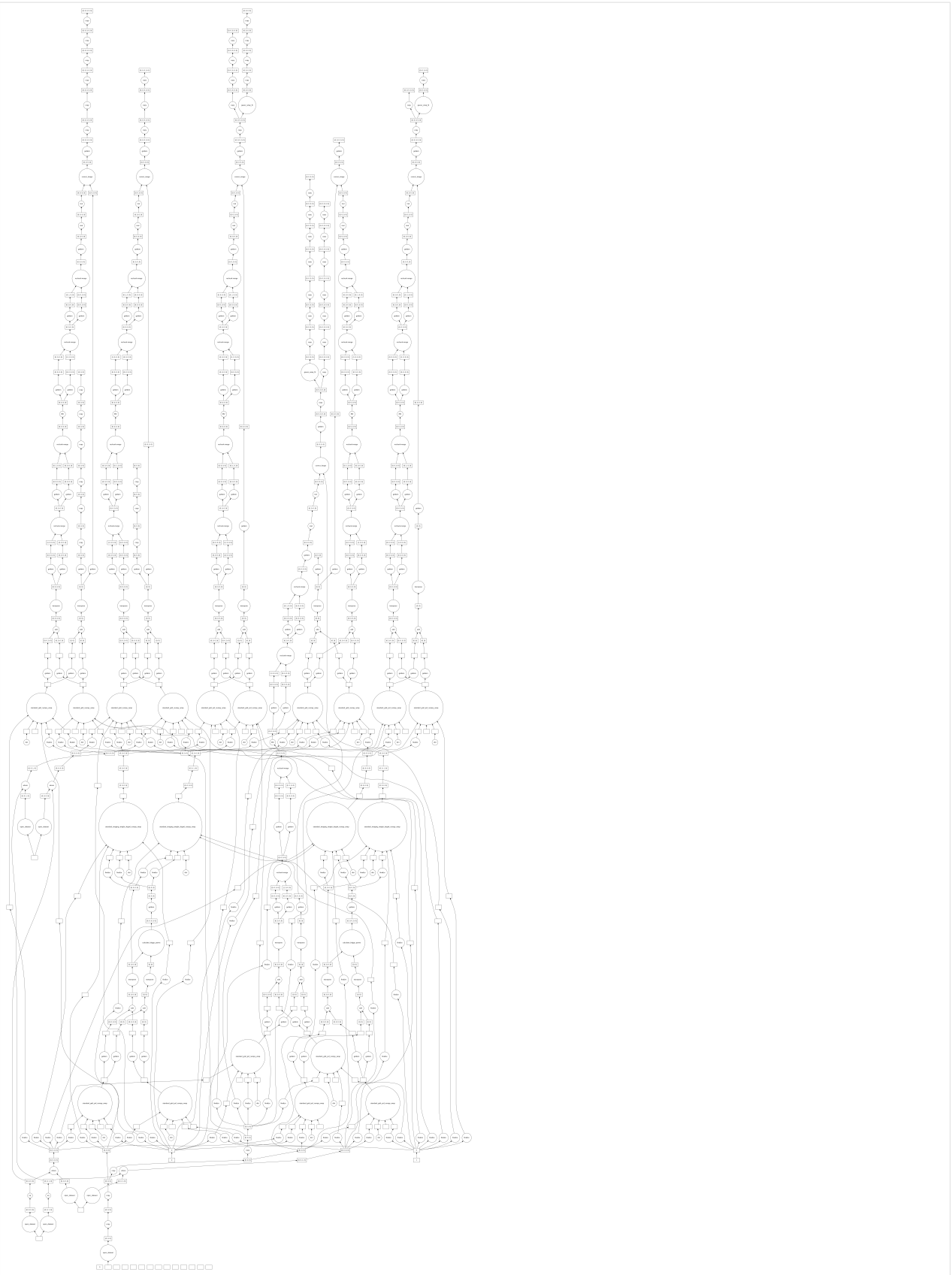
(continues on next page)

(continued from previous page)

```
    PSF_BRG          (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 200,␣
↪1, 1, 1), meta=np.ndarray>
    PSF_FIT_BRG      (time, chan, pol, elps_index) float64 dask.array<chunksize=(1, 1,
↪ 1, 3), meta=np.ndarray>
    IMAGE_BRG        (l, m, time, chan, pol) float64 dask.array<chunksize=(200, 200,␣
↪1, 1, 1), meta=np.ndarray>
Attributes:
    data_groups:          [{'0': {'id': '0', 'sum_weight': 'SUM_WEIGHT_NAT', ...
    axis_units:           ['rad', 'rad', 'time', 'Hz', 'pol']
    direction_reference:  FK5
    spectral_reference:   lsrk
    velocity_type:        radio
    unit:                 Jy/beam
```

[5]:
```
dask.visualize(img_xds,'weighting_image_graph.png')
```

[5]:



8.3.5 Save To Disk

[6]:
```
from cngi.dio import write_image
img_xds = write_image(img_xds, outfile='twhya_standard_gridder_lsrk_mfs.img.zarr')
```

```
Time to store and execute graph  write_zarr 35.28575277328491
```

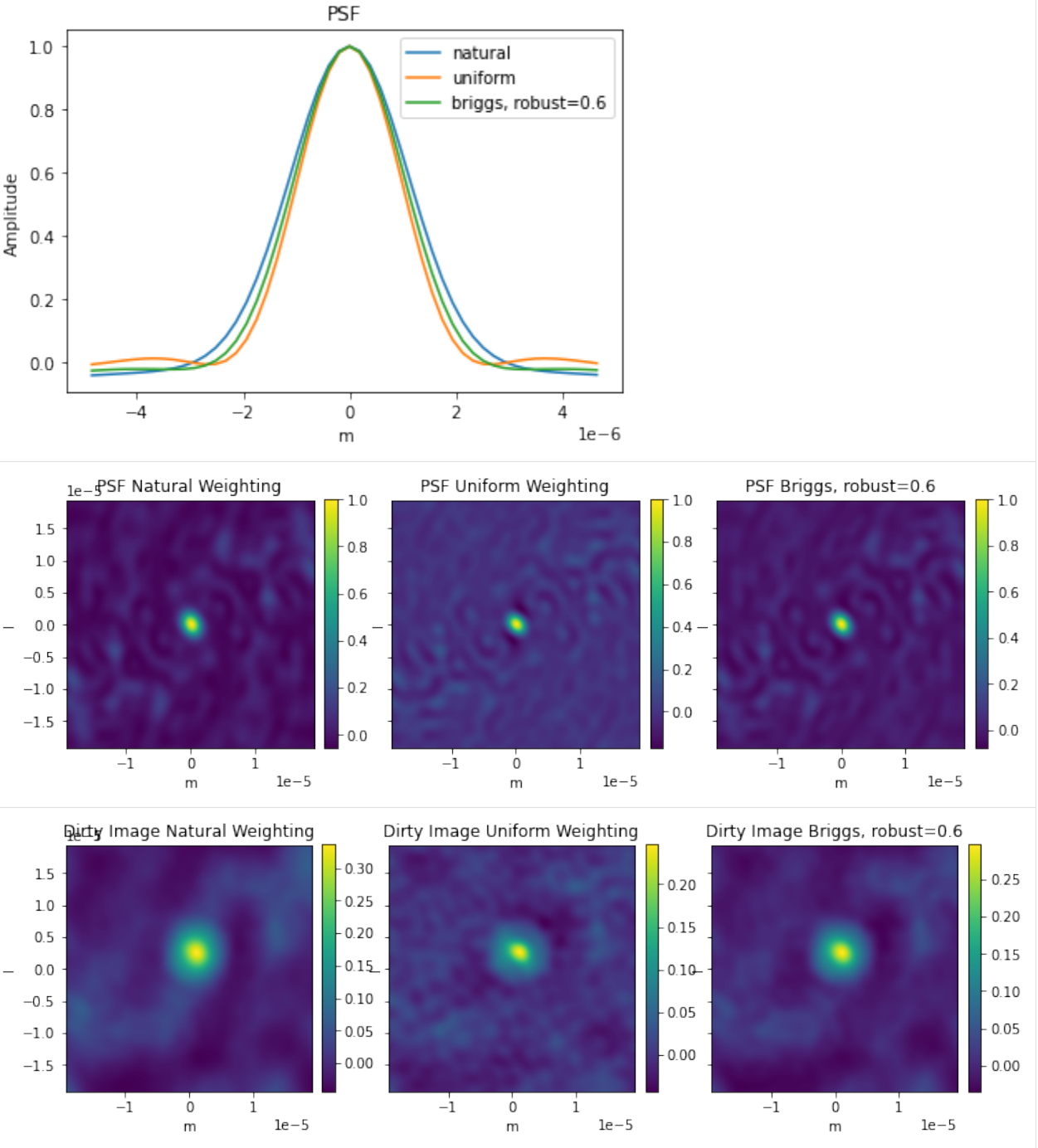
8.3.6 Plot

[7]:
```
import xarray as xr
import matplotlib.pylab as plt
img_xds = xr.open_zarr('twhya_standard_gridder_lsrk_mfs.img.zarr').isel(time=0,chan=0,
↪pol=0)
import numpy as np
plt.close('all')

## Plot PSF images
plt.figure()
plt.plot(img_xds.PSF_NAT[100,75:125].m,img_xds.PSF_NAT[100,75:125],label='natural')
plt.plot(img_xds.PSF_NAT[100,75:125].m,img_xds.PSF_UNI[100,75:125],label='uniform')
plt.plot(img_xds.PSF_NAT[100,75:125].m,img_xds.PSF_BRG[100,75:125],label='briggs,␣
↪robust='+str(imaging_weights_parms['robust']))
plt.legend()
plt.title('PSF')
plt.xlabel('m')
plt.ylabel('Amplitude')
plt.show()

extent = extent=(np.min(img_xds.m),np.max(img_xds.m),np.min(img_xds.l),np.max(img_xds.
↪l))

fig0, ax0 = plt.subplots(1, 3, sharey=True,figsize=(12, 5))
im0 = ax0[0].imshow(img_xds.PSF_NAT,extent=extent)
im1 = ax0[1].imshow(img_xds.PSF_UNI,extent=extent)
im2 = ax0[2].imshow(img_xds.PSF_BRG,extent=extent)
ax0[0].title.set_text('PSF Natural Weighting')
ax0[1].title.set_text('PSF Uniform Weighting')
ax0[2].title.set_text('PSF Briggs, robust='+str(imaging_weights_parms['robust']))
ax0[0].set_xlabel('m'),ax0[1].set_xlabel('m'),ax0[0].set_ylabel('l'),ax0[1].set_
↪ylabel('l'),ax0[2].set_xlabel('m'),ax0[2].set_ylabel('l')
fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
fig0.colorbar(im2, ax=ax0[2], fraction=0.046, pad=0.04)
plt.show()

fig0, ax0 = plt.subplots(1, 3, sharey=True,figsize=(12, 5))
im0 = ax0[0].imshow(img_xds.IMAGE_NAT,extent=extent)
im1 = ax0[1].imshow(img_xds.IMAGE_UNI,extent=extent)
im2 = ax0[2].imshow(img_xds.IMAGE_BRG,extent=extent)
ax0[0].title.set_text('Dirty Image Natural Weighting')
ax0[1].title.set_text('Dirty Image Uniform Weighting')
ax0[2].title.set_text('Dirty Image Briggs, robust='+str(imaging_weights_parms['robust
↪']))
ax0[0].set_xlabel('m'),ax0[1].set_xlabel('m'),ax0[0].set_ylabel('l'),ax0[1].set_
↪ylabel('l'),ax0[2].set_xlabel('m'),ax0[2].set_ylabel('l')
fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
```

(continues on next page)

(continued from previous page)

```
fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)
fig0.colorbar(im2, ax=ax0[2], fraction=0.046, pad=0.04)
plt.show()
```



Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/imaging/mosaic_image_example.ipynb

8.4 Mosaic Imaging

This notebook will demonstrate how to create a joint mosaic image. Data is taken from all the pointings in an input visibility dataset and combined to have a single phase-reference center.

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here.

8.4.1 Installation

[1]:
```
import os
os.system("pip install cngi-prototype==0.0.91")
print('complete')
```

```
complete
```

8.4.2 Dataset

The simulated dataset consists of three fields which contain four point sources over three frequency channels. The ALMA layout `alma.cycle6.3.cfg` is used, which can be found here.

[2]:
```
!gdown -q --id 1KzWk0Xg8-xpljTL6m8WEE8KbfLTpskRg
!unzip alma12m_3field_dovpTrue.vis.zarr.zip > /dev/null

!gdown -q --id 1YdJGBi2qtdCuJ6dm4xrXU5zvXeJc7w3v
!unzip alma12m_3field_dovpTrue_gridder_mosaic.img.zarr.zip > /dev/null

#%matplotlib widget
```

8.4.3 Load Dataset

[3]:
```
import xarray as xr
from cngi.dio import read_vis

xr.set_options(display_style="html")

infile = "alma12m_3field_dovpTrue.vis.zarr"
mxds = read_vis(infile)
mxds
```

```
overwrite_encoded_chunks True
```

[3]:
```
<xarray.Dataset>
Dimensions:           (antenna_ids: 43, feed_ids: 43, field_ids: 3, observation_ids:␣
↪1, polarization_ids: 1, source_ids: 3, spw_ids: 1, state_ids: 1)
Coordinates:
  * antenna_ids       (antenna_ids) int64 0 1 2 3 4 5 6 ... 36 37 38 39 40 41 42
    antennas          (antenna_ids) <U16 'A001' 'A002' 'A007' ... 'A085' 'A088'
  * field_ids         (field_ids) int64 0 1 2
    fields            (field_ids) <U16 'field_1' 'field_2' 'field_3'
  * feed_ids          (feed_ids) int32 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  * observation_ids   (observation_ids) int64 0
    observations      (observation_ids) <U16 'CASA simulation'
```

(continues on next page)

(continued from previous page)

```
  * polarization_ids  (polarization_ids) int64 0
  * source_ids        (source_ids) int32 0 1 2
    sources           (source_ids) <U16 'field_1' 'field_2' 'field_3'
  * spw_ids           (spw_ids) int64 0
  * state_ids         (state_ids) int64 0
Data variables:
    *empty*
Attributes:
    xds0:             <xarray.Dataset>\nDimensions:                (baseline:...
    ANTENNA:          <xarray.Dataset>\nDimensions:        (antenna_id: 43, d...
    FEED:             <xarray.Dataset>\nDimensions:             (d0: 43, d1: ...
    FIELD:            <xarray.Dataset>\nDimensions:        (d1: 1, d2: 2, fie...
    OBSERVATION:      <xarray.Dataset>\nDimensions:         (d1: 2, observati...
    POINTING:         <xarray.Dataset>\nDimensions:      (antenna_id: 43, d2:...
    POLARIZATION:     <xarray.Dataset>\nDimensions:       (d0: 1, d1: 1, d2: ...
    SOURCE:           <xarray.Dataset>\nDimensions:             (d0: 3, d1: 2...
    SPECTRAL_WINDOW:  <xarray.Dataset>\nDimensions:             (d1: 3, spect...
    STATE:            <xarray.Dataset>\nDimensions:   (state_id: 1)\nCoordina...
```

[4]:
```
mxds.xds0
```

[4]:
```
<xarray.Dataset>
Dimensions:                (baseline: 903, chan: 3, pol: 1, pol_id: 1, spw_id: 1,␣
↪time: 192, uvw_index: 3)
Coordinates:
  * baseline               (baseline) int64 0 1 2 3 4 5 ... 898 899 900 901 902
  * chan                   (chan) float64 3.4e+11 3.74e+11 4.08e+11
    chan_width             (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    effective_bw           (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * pol                    (pol) int32 9
  * pol_id                 (pol_id) int32 0
    resolution             (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * spw_id                 (spw_id) int32 0
  * time                   (time) datetime64[ns] 2011-05-27T04:32:14.77260398...
Dimensions without coordinates: uvw_index
Data variables: (12/20)
    ANTENNA1               (baseline) int32 dask.array<chunksize=(301,), meta=np.
↪ndarray>
    ANTENNA2               (baseline) int32 dask.array<chunksize=(301,), meta=np.
↪ndarray>
    ARRAY_ID               (time, baseline) int32 dask.array<chunksize=(32, 301),␣
↪meta=np.ndarray>
    CORRECTED_DATA         (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(32, 301, 1, 1), meta=np.ndarray>
    CORRECTED_DATA_WEIGHT  (time, baseline, chan, pol) float64 dask.array
↪<chunksize=(32, 301, 1, 1), meta=np.ndarray>
    DATA                   (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(32, 301, 1, 1), meta=np.ndarray>
    ...                     ...
    OBSERVATION_ID         (time, baseline) int32 dask.array<chunksize=(32, 301),␣
↪meta=np.ndarray>
    PROCESSOR_ID           (time, baseline) int32 dask.array<chunksize=(32, 301),␣
↪meta=np.ndarray>
    SCAN_NUMBER            (time, baseline) int32 dask.array<chunksize=(32, 301),␣
↪meta=np.ndarray>
    STATE_ID               (time, baseline) int32 dask.array<chunksize=(32, 301),␣
↪meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    TIME_CENTROID          (time, baseline) float64 dask.array<chunksize=(32, 301),␣
↪meta=np.ndarray>
    UVW                    (time, baseline, uvw_index) float64 dask.array
↪<chunksize=(32, 301, 3), meta=np.ndarray>
Attributes:
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:  Group 1
    if_conv_chain:    0
    meas_freq_ref:    1
    name:             Band_8
    net_sideband:     1
    num_chan:         3
    num_corr:         1
    ref_frequency:    340000000000.0
    total_bandwidth:  101999999999.99998
```

8.4.4 Grid Parameters

[5]:
```
grid_parms = {}
grid_parms['chan_mode'] = 'cube'
grid_parms['image_size'] = [1000,720]
grid_parms['cell_size'] = [0.04,0.04]
grid_parms['fft_padding'] = 1.0
grid_parms['phase_center'] = mxds.FIELD.PHASE_DIR[1,0,:].data.compute()
```

8.4.5 Direction Rotation

The UVW coordinates must be rotated and the visbility DATA must be phase rotated, relative to the mosaic phase center specified by `rotation_parms['image_phase_center']`.

direction_rotate documentation

[6]:
```
from ngcasa.imaging import direction_rotate
import numpy as np
import dask

xr.set_options(display_style="html")

infile = "alma12m_3field_dovpTrue.vis.zarr"
#DATA shape (192, 903, 3, 1), ZARR chunking (32, 301, 1, 1)
mxds = read_vis(infile,chunks={'time':192,'baseline':903,'chan':1})

sel_parms = {}
sel_parms['xds'] = 'xds0'

rotation_parms = {}
rotation_parms['new_phase_center'] = grid_parms['phase_center']
rotation_parms['common_tangent_reprojection'] = True
rotation_parms['single_precision'] = False

mxds = direction_rotate(mxds, rotation_parms, sel_parms)
```

(continues on next page)

(continued from previous page)

```
mxds.xds0
```

```
overwrite_encoded_chunks True
######################### Start direction_rotate ##########################
Setting default data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw':
↪'UVW', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT'}
######################### Created graph for direction_rotate ########################
```

[6]:
```
<xarray.Dataset>
Dimensions:                (baseline: 903, chan: 3, pol: 1, pol_id: 1, spw_id: 1,␣
↪time: 192, uvw_index: 3)
Coordinates:
  * baseline               (baseline) int64 0 1 2 3 4 5 ... 898 899 900 901 902
  * chan                   (chan) float64 3.4e+11 3.74e+11 4.08e+11
  * pol                    (pol) int32 9
  * time                   (time) datetime64[ns] 2011-05-27T04:32:14.77260398...
    chan_width             (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    effective_bw           (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * pol_id                 (pol_id) int32 0
    resolution             (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * spw_id                 (spw_id) int32 0
Dimensions without coordinates: uvw_index
Data variables: (12/22)
    ANTENNA1               (baseline) int32 dask.array<chunksize=(903,), meta=np.
↪ndarray>
    ANTENNA2               (baseline) int32 dask.array<chunksize=(903,), meta=np.
↪ndarray>
    ARRAY_ID               (time, baseline) int32 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    CORRECTED_DATA         (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
    CORRECTED_DATA_WEIGHT  (time, baseline, chan, pol) float64 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
    DATA                   (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
    ...                     ...
    SCAN_NUMBER            (time, baseline) int32 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    STATE_ID               (time, baseline) int32 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    TIME_CENTROID          (time, baseline) float64 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    UVW                    (time, baseline, uvw_index) float64 dask.array
↪<chunksize=(192, 903, 3), meta=np.ndarray>
    UVW_ROT                (time, baseline, uvw_index) float64 dask.array
↪<chunksize=(192, 903, 3), meta=np.ndarray>
    DATA_ROT               (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
Attributes:
    corr_product:      [[0, 0]]
    data_groups:       [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:        0
    freq_group_name:   Group 1
    if_conv_chain:     0
    meas_freq_ref:     1
```

(continues on next page)

(continued from previous page)

```
    name:             Band_8
    net_sideband:     1
    num_chan:         3
    num_corr:         1
    ref_frequency:    340000000000.0
    total_bandwidth:  101999999999.99998
```

8.4.6 Make Imaging Weights

make_imaging_weight documentation

[7]:
```
from ngcasa.imaging import make_imaging_weight

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 2

mxds = make_imaging_weight(mxds, imaging_weights_parms, grid_parms, sel_parms)

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0

mxds = make_imaging_weight(mxds, imaging_weights_parms, grid_parms, sel_parms)
mxds.xds0
```

```
######################### Start make_imaging_weights #########################
Setting data_group_in  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'IMAGING_WEIGHT'}
Since weighting is natural input weight will be reused as imaging weight.
######################### Created graph for make_imaging_weight ######################
↪###
######################### Start make_imaging_weights #########################
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw':
↪ 'UVW', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'IMAGING_WEIGHT'}
Since weighting is natural input weight will be reused as imaging weight.
######################### Created graph for make_imaging_weight ######################
↪###
```

[7]:
```
<xarray.Dataset>
Dimensions:                (baseline: 903, chan: 3, pol: 1, pol_id: 1, spw_id: 1,␣
↪time: 192, uvw_index: 3)
Coordinates:
  * baseline               (baseline) int64 0 1 2 3 4 5 ... 898 899 900 901 902
  * chan                   (chan) float64 3.4e+11 3.74e+11 4.08e+11
  * pol                    (pol) int32 9
```

(continues on next page)

(continued from previous page)

```
  * time                    (time) datetime64[ns] 2011-05-27T04:32:14.77260398...
    chan_width              (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    effective_bw            (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * pol_id                  (pol_id) int32 0
    resolution              (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
  * spw_id                  (spw_id) int32 0
Dimensions without coordinates: uvw_index
Data variables: (12/22)
    ANTENNA1                (baseline) int32 dask.array<chunksize=(903,), meta=np.
↪ndarray>
    ANTENNA2                (baseline) int32 dask.array<chunksize=(903,), meta=np.
↪ndarray>
    ARRAY_ID                (time, baseline) int32 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    CORRECTED_DATA          (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
    CORRECTED_DATA_WEIGHT   (time, baseline, chan, pol) float64 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
    DATA                    (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
    ...                      ...
    SCAN_NUMBER             (time, baseline) int32 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    STATE_ID                (time, baseline) int32 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    TIME_CENTROID           (time, baseline) float64 dask.array<chunksize=(192, 903),␣
↪meta=np.ndarray>
    UVW                     (time, baseline, uvw_index) float64 dask.array
↪<chunksize=(192, 903, 3), meta=np.ndarray>
    UVW_ROT                 (time, baseline, uvw_index) float64 dask.array
↪<chunksize=(192, 903, 3), meta=np.ndarray>
    DATA_ROT                (time, baseline, chan, pol) complex128 dask.array
↪<chunksize=(192, 903, 1, 1), meta=np.ndarray>
Attributes:
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:  Group 1
    if_conv_chain:    0
    meas_freq_ref:    1
    name:             Band_8
    net_sideband:     1
    num_chan:         3
    num_corr:         1
    ref_frequency:    340000000000.0
    total_bandwidth:  101999999999.99998
```

8.4.7 Make Gridding Convolution Functions

make_gridding_convolution_function

[8]:
```
from ngcasa.imaging import make_gridding_convolution_function
import numpy as np
import dask.array as da
from cngi.dio import write_image

gcf_parms = {}
gcf_parms['function'] = 'casa_airy'
gcf_parms['list_dish_diameters'] = np.array([10.7])
gcf_parms['list_blockage_diameters'] = np.array([0.75])

unique_ant_indx = mxds.ANTENNA.DISH_DIAMETER.values
unique_ant_indx[unique_ant_indx == 12.0] = 0

gcf_parms['unique_ant_indx'] = unique_ant_indx.astype(int)
gcf_parms['phase_center'] = grid_parms['phase_center']

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 2

gcf_xds = make_gridding_convolution_function(mxds, gcf_parms, grid_parms, sel_parms)
write_image(gcf_xds,'mosaic_gcf.gcf.zarr')
gcf_xds = xr.open_zarr('mosaic_gcf.gcf.zarr')
gcf_xds
```

```
######################### Start make_gridding_convolution_function ##################
↪######
Setting data_group_in  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '3',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default chan_tolerance_factor  to  0.005
Setting default oversampling  to  [10, 10]
Setting default max_support  to  [15, 15]
Setting default support_cut_level  to  0.025
Setting default a_chan_num_chunk  to  3
Setting default image_center  to  [500 360]
#########################  Created graph for make_gridding_convolution_function ######
↪###################
Time to store and execute graph  write_zarr 7.4372382164001465
```

[8]:
```
<xarray.Dataset>
Dimensions:              (baseline: 903, chan: 3, conv_baseline: 1, conv_chan: 3, conv_
↪pol: 1, field_id: 3, l: 1000, m: 720, pol: 1, u: 160, v: 160, xy: 2)
Coordinates:
  * field_id             (field_id) int64 0 1 2
  * l                    (l) int64 0 1 2 3 4 5 6 ... 993 994 995 996 997 998 999
  * m                    (m) int64 0 1 2 3 4 5 6 ... 713 714 715 716 717 718 719
  * u                    (u) int64 0 1 2 3 4 5 6 ... 153 154 155 156 157 158 159
  * v                    (v) int64 0 1 2 3 4 5 6 ... 153 154 155 156 157 158 159
  * xy                   (xy) int64 0 1
Dimensions without coordinates: baseline, chan, conv_baseline, conv_chan, conv_pol,␣
↪pol
Data variables:
    CF_BASELINE_MAP      (baseline) int64 dask.array<chunksize=(903,), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    CF_CHAN_MAP          (chan) int64 dask.array<chunksize=(1,), meta=np.ndarray>
    CF_POL_MAP           (pol) int64 dask.array<chunksize=(1,), meta=np.ndarray>
    CONV_KERNEL          (conv_baseline, conv_chan, conv_pol, u, v) float64 dask.array
↪<chunksize=(1, 1, 1, 160, 160), meta=np.ndarray>
    PHASE_GRADIENT       (field_id, u, v) complex128 dask.array<chunksize=(1, 160,␣
↪160), meta=np.ndarray>
    PS_CORR_IMAGE        (l, m) float64 dask.array<chunksize=(1000, 720), meta=np.
↪ndarray>
    SUPPORT              (conv_baseline, conv_chan, conv_pol, xy) int64 dask.array
↪<chunksize=(1, 1, 1, 2), meta=np.ndarray>
    WEIGHT_CONV_KERNEL   (conv_baseline, conv_chan, conv_pol, u, v) float64 dask.array
↪<chunksize=(1, 1, 1, 160, 160), meta=np.ndarray>
Attributes:
    cell_uv:           [-515.6620156177408, 716.197243913529]
    oversampling:      [10, 10]
    write_zarr_time:   7.4372382164001465
```

8.4.8 Make Mosaic Primary Beam, PSF, and Image

make_mosaic_pb

make_psf

make_image_with_gcf

[9]:
```
from ngcasa.imaging import make_mosaic_pb
from cngi.dio import read_image

vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 2

img_sel_parms = {}
img_xds= xr.Dataset()

img_xds = make_mosaic_pb(mxds,gcf_xds,img_xds,vis_sel_parms,img_sel_parms,grid_parms)

################################################

from ngcasa.imaging import make_psf

vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 2

img_sel_parms = {}
img_sel_parms['data_group_out_id'] = 0

img_xds = make_psf(mxds, img_xds, grid_parms, vis_sel_parms, img_sel_parms)

################################################

from ngcasa.imaging import make_image_with_gcf
from cngi.dio import write_image

vis_select_parms = {}
```

(continues on next page)

(continued from previous page)

```
vis_select_parms['xds'] = 'xds0'
vis_select_parms['data_group_in_id'] = 2

img_select_parms = {}
img_select_parms['data_group_in_id'] = 0
img_select_parms['data_group_out_id'] = 0

norm_parms = {}
norm_parms['norm_type'] = 'flat_sky'

img_xds = make_image_with_gcf(mxds,gcf_xds, img_xds, grid_parms, norm_parms, vis_
↪select_parms, img_select_parms)
write_image(img_xds,'mosaic_img.img.zarr')
img_xds = read_image('mosaic_img.img.zarr')
img_xds
```

```
######################### Start make_mosaic_pb #########################
Setting default image_center to [500 360]
Setting data_group_in to {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out to {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '3',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_in to {'id': '0'}
Setting default data_group_out to {'id': '0', 'pb': 'PB', 'weight_pb': 'WEIGHT_PB',
↪'weight_pb_sum_weight': 'WEIGHT_PB_SUM_WEIGHT'}
#########################  Created graph for make_mosaic_pb ########################
######################### Start make_psf #########################
Setting default image_center to [500 360]
Setting data_group_in to {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out to {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '3',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_in to {'id': '0', 'pb': 'PB', 'weight_pb': 'WEIGHT_PB',
↪'weight_pb_sum_weight': 'WEIGHT_PB_SUM_WEIGHT'}
Setting default data_group_out [' pb '] to PB
Setting default data_group_out [' weight_pb '] to WEIGHT_PB
Setting default data_group_out [' weight_pb_sum_weight '] to WEIGHT_PB_SUM_WEIGHT
Setting default data_group_out [' sum_weight '] to PSF_SUM_WEIGHT
Setting default data_group_out [' psf '] to PSF
Setting default data_group_out [' psf_fit '] to PSF_FIT
######################### Created graph for make_psf #########################
######################### Start make_image_with_gcf #########################
Setting default image_center to [500 360]
Setting default single_precision to True
Setting default pb_limit to 0.2
Setting data_group_in to {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out to {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '3',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting data_group_in to {'id': '0', 'pb': 'PB', 'weight_pb': 'WEIGHT_PB', 'weight_
↪pb_sum_weight': 'WEIGHT_PB_SUM_WEIGHT', 'sum_weight': 'PSF_SUM_WEIGHT', 'psf': 'PSF
↪', 'psf_fit': 'PSF_FIT'}
Setting default data_group_out [' pb '] to PB
Setting default data_group_out [' weight_pb '] to WEIGHT_PB
Setting default data_group_out [' weight_pb_sum_weight '] to WEIGHT_PB_SUM_WEIGHT
Setting default data_group_out [' sum_weight '] to SUM_WEIGHT
Setting default data_group_out [' psf '] to PSF
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_out [' psf_fit ']  to  PSF_FIT
Setting default data_group_out [' image ']  to  IMAGE
grid sizes 1000 720
#########################  Created graph for make_mosaic_with_gcf ###################
↪#####
Time to store and execute graph  write_zarr 10.509612560272217
```

[9]:
```
<xarray.Dataset>
Dimensions:                (chan: 3, elps_index: 3, l: 1000, m: 720, pol: 1, time: 1)
Coordinates:
  * chan                   (chan) float64 3.4e+11 3.74e+11 4.08e+11
    chan_width             (chan) float64 dask.array<chunksize=(1,), meta=np.ndarray>
    declination            (l, m) float64 dask.array<chunksize=(250, 180), meta=np.
↪ndarray>
  * l                      (l) float64 9.696e-05 9.677e-05 ... -9.677e-05
  * m                      (m) float64 -6.981e-05 -6.962e-05 ... 6.962e-05
  * pol                    (pol) int32 9
    right_ascension        (l, m) float64 dask.array<chunksize=(250, 180), meta=np.
↪ndarray>
  * time                   (time) datetime64[ns] 2011-05-27T20:27:14.772603989
Dimensions without coordinates: elps_index
Data variables:
    IMAGE                  (l, m, time, chan, pol) float64 dask.array<chunksize=(1000,␣
↪720, 1, 1, 1), meta=np.ndarray>
    PB                     (l, m, time, chan, pol) float64 dask.array<chunksize=(1000,␣
↪720, 1, 1, 1), meta=np.ndarray>
    PSF                    (l, m, time, chan, pol) float64 dask.array<chunksize=(1000,␣
↪720, 1, 1, 1), meta=np.ndarray>
    PSF_FIT                (time, chan, pol, elps_index) float64 dask.array
↪<chunksize=(1, 1, 1, 3), meta=np.ndarray>
    PSF_SUM_WEIGHT         (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    SUM_WEIGHT             (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
    WEIGHT_PB              (l, m, time, chan, pol) float64 dask.array<chunksize=(1000,␣
↪720, 1, 1, 1), meta=np.ndarray>
    WEIGHT_PB_SUM_WEIGHT   (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes:
    axis_units:            ['rad', 'rad', 'time', 'Hz', 'pol']
    data_groups:           [{'0': {'id': '0', 'image': 'IMAGE', 'pb': 'PB', 'p...
    direction_reference:   FK5
    spectral_reference:    lsrk
    unit:                  Jy/beam
    velocity_type:         radio
    write_zarr_time:       10.509612560272217
```

8.4.9 Compare CASA and ngCASA Primary Beams

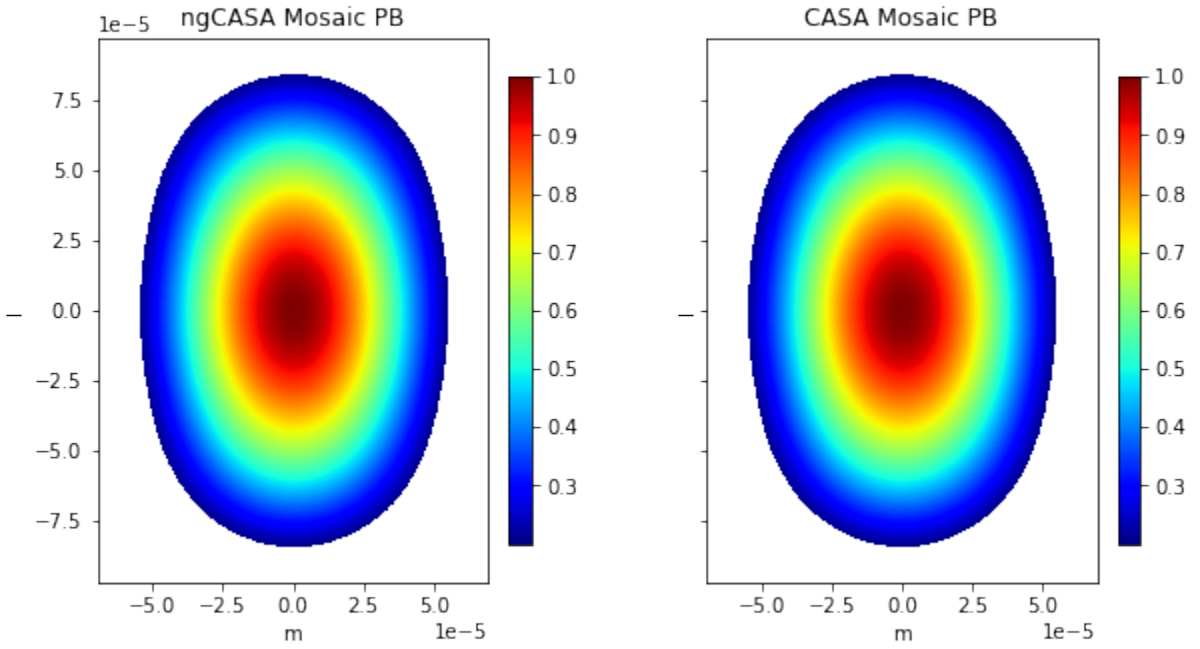
[10]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive
import scipy
from scipy.signal import decimate
from cngi.image import implot

img_xds = read_image('mosaic_img.img.zarr').isel(time=0,pol=0)
casa_img_xds = read_image('alma12m_3field_dovpTrue_gridder_mosaic.img.zarr').
↪isel(time=0,pol=0)
pb_limit = 0.2
extent = extent=(np.min(casa_img_xds.m),np.max(casa_img_xds.m),np.min(casa_img_xds.l),
↪np.max(casa_img_xds.l))

def comparison_plots(chan):
    plt.close('all')
    print('Frequency',img_xds.chan[chan].values/10**9, 'GHz')
    mosaic_pb = img_xds.PB.isel(chan=chan)
    mosaic_pb = mosaic_pb.where(mosaic_pb > pb_limit,other=np.nan)

    casa_mosaic_pb = casa_img_xds.PB.isel(chan=chan)
    casa_mosaic_pb = casa_mosaic_pb.where(casa_mosaic_pb > pb_limit,other=np.nan)

    fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
    im0 = ax0[0].imshow(mosaic_pb,extent=extent,cmap='jet')
    im1 = ax0[1].imshow(casa_mosaic_pb,extent=extent,cmap='jet')
    ax0[0].title.set_text('ngCASA Mosaic PB')
    ax0[1].title.set_text('CASA Mosaic PB')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'), ax0[1].
↪set_ylabel('l')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)

    plt.figure()
    plt.plot(casa_mosaic_pb.l,mosaic_pb.isel(m=360),label='ngCASA PB')
    plt.plot(casa_mosaic_pb.l,casa_mosaic_pb.isel(m=360),'*',label='CASA PB',
↪markersize=1)
    plt.legend()
    plt.xlabel('l')
    plt.ylabel('Amplitude')
    plt.title('Mosaic PB Cross Section')

    diff_image = mosaic_pb - casa_mosaic_pb

    plt.figure()
    plt.imshow(100*(mosaic_pb - casa_mosaic_pb),extent=extent,cmap='jet')
    plt.xlabel('m'), plt.ylabel('l')
    plt.colorbar()
    plt.title('Percentage Difference Mosaic PB')

    plt.show()

interactive_plot = interactive(comparison_plots, chan=(0, 2))
output = interactive_plot.children[-1]
output.layout.auto_scroll_threshold = 9999;
```

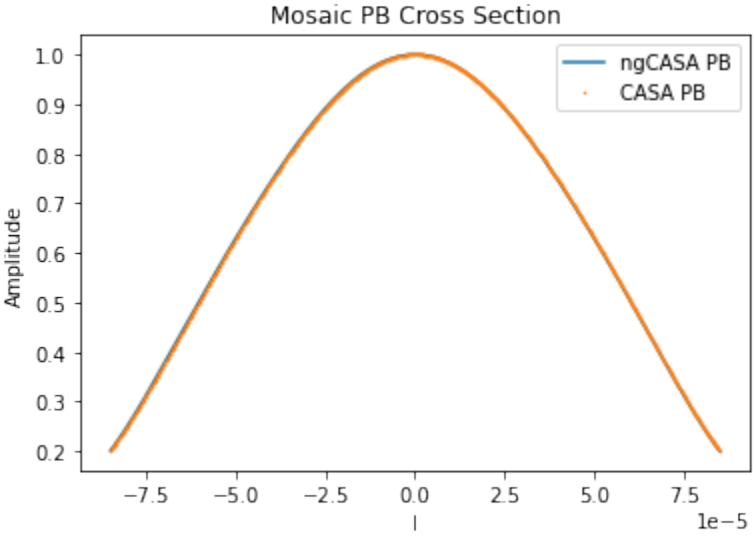
(continues on next page)

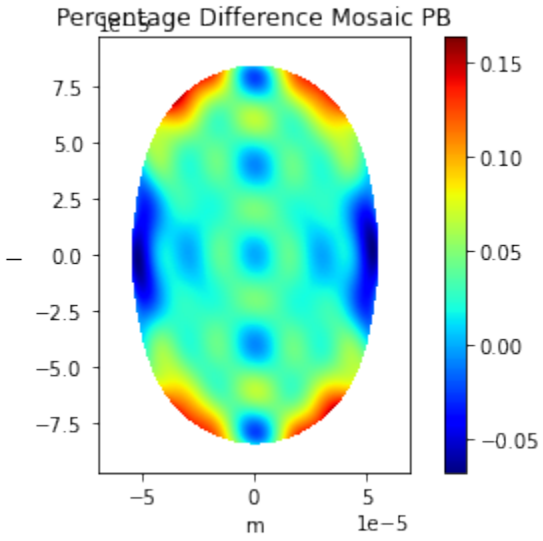
(continued from previous page)

```
#interactive_plot
comparison_plots(1)
```

```
Frequency 374.0 GHz
```







8.4.10 Get Simulated Sources l,m Coordinates

[11]:
```
from astropy.coordinates import SkyCoord
from astropy.wcs import WCS
rad_to_deg =  180/np.pi
deg_to_rad = np.pi/180
arcsec_to_deg = 1/3600
arcsec_to_rad = np.pi/(180*3600)

phase_center = grid_parms['phase_center']
w = WCS(naxis=2)
w.wcs.crpix = np.array(grid_parms['image_size'])//2
w.wcs.cdelt = np.array([-0.04,0.04])*arcsec_to_deg
w.wcs.crval = phase_center*rad_to_deg
w.wcs.ctype = ['RA---SIN','DEC--SIN']

ra = ['12h01m51.903005s','12h01m52.430856s','12h01m52.958707s','12h01m52.259s',
↪'12h01m52s','12h01m53.153s']
dec = ['-18d51m49.94373s','-18d51m49.94369s','-18d51m49.94365s','-18d51m42.983s','-
↪18d51m46s','-18d51m59.305s']
ps_skycoord = SkyCoord(ra=ra,dec=dec,frame='fk5')


ra_dec = np.array([ps_skycoord.ra.degree,ps_skycoord.dec.degree]).T
lm_pix_pos = w.all_world2pix(ra_dec, 1)

cell_size = np.array(grid_parms['cell_size'])*arcsec_to_rad
cell_size[0] = -cell_size[0]
image_center = np.array(grid_parms['image_size'])//2
source_lm_pos = lm_pix_pos*cell_size - image_center*cell_size
```

8.4.11 Compare CASA and ngCASA Sky Images

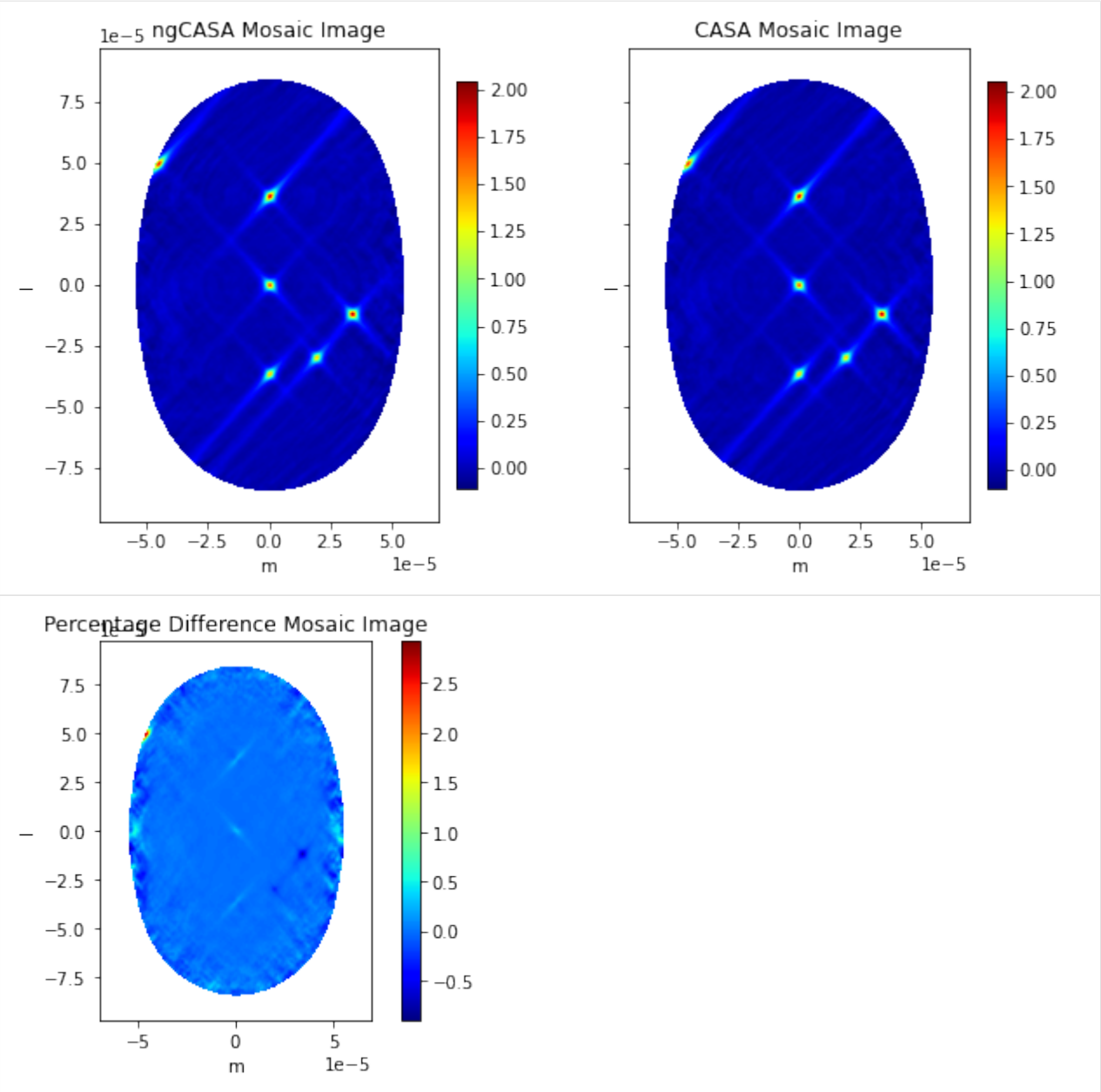
[12]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive
import scipy
from scipy.signal import decimate

img_xds = read_image('mosaic_img.img.zarr',chunks={'l':grid_parms['image_size'][0],'m
↪':grid_parms['image_size'][1]}).isel(time=0,pol=0)
casa_img_xds = read_image('alma12m_3field_dovpTrue_gridder_mosaic.img.zarr',chunks={'l
↪':grid_parms['image_size'][0],'m':grid_parms['image_size'][1]}).isel(time=0,pol=0)
pb_limit = 0.2
extent = extent=(np.min(casa_img_xds.m),np.max(casa_img_xds.m),np.min(casa_img_xds.l),
↪np.max(casa_img_xds.l))

ngcasa_image_name = 'IMAGE'
casa_image_name = 'IMAGE_PBCOR'

def comparison_plots(chan):
    print('Frequency',img_xds.chan[chan].values/10**9, 'GHz')
    mosaic_pb = img_xds.PB.isel(chan=chan)
    casa_mosaic_pb = casa_img_xds.PB.isel(chan=chan)

    mosaic_img = img_xds[ngcasa_image_name].isel(chan=chan)
    mosaic_img = mosaic_img.where(mosaic_pb > pb_limit,other=np.nan)

    casa_mosaic_img = casa_img_xds[casa_image_name].isel(chan=chan)
    casa_mosaic_img = casa_mosaic_img.where(casa_mosaic_pb > pb_limit,other=np.nan)

    sim_sources = np.array([1.5,1.76,2.0,2.0,1.456,1.888])

    print('################## Flux of Point Sources ##################')
    print('Sim    ','ngCASA  ', 'CASA')
    for i,s in enumerate(sim_sources):
        ngcasa_recovered_val = img_xds[ngcasa_image_name].isel(chan=chan).
↪interp(l=source_lm_pos[i,0],m=source_lm_pos[i,1]).values
        casa_recovered_val = casa_mosaic_img.interp(l=source_lm_pos[i,0],m=source_lm_
↪pos[i,1]).values
        print('{0:.3f}  '.format(s),'{0:.4f}  '.format(ngcasa_recovered_val),'{0:.4f}
↪'.format(casa_recovered_val))


    print('############ Percentage Difference Flux to Sim ############')
    print('ngCASA  ', 'CASA')
    for i,s in enumerate(sim_sources):
        ngcasa_recovered_val = img_xds[ngcasa_image_name].isel(chan=chan).
↪interp(l=source_lm_pos[i,0],m=source_lm_pos[i,1]).values
        casa_recovered_val = casa_mosaic_img.interp(l=source_lm_pos[i,0],m=source_lm_
↪pos[i,1]).values
        print('{0:.4f}  '.format(100*(s-ngcasa_recovered_val)/s),'{0:.4f}'.
↪format(100*(s-casa_recovered_val)/s))

    fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
    im0 = ax0[0].imshow(mosaic_img,cmap='jet',extent=extent)
    im1 = ax0[1].imshow(casa_mosaic_img,cmap='jet',extent=extent)
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'), ax0[1].
↪set_ylabel('l')
```

(continues on next page)

(continued from previous page)

```
    ax0[0].title.set_text('ngCASA Mosaic Image')
    ax0[1].title.set_text('CASA Mosaic Image')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)

    plt.figure()
    plt.imshow((100*(mosaic_img - casa_mosaic_img)/2),cmap='jet',extent=extent)
    plt.colorbar()
    plt.xlabel('m'), plt.ylabel('l')
    plt.title('Percentage Difference Mosaic Image')

    plt.show()

interactive_plot = interactive(comparison_plots, chan=(0, 2))
output = interactive_plot.children[-1]
output.layout.height = '1450px'
#interactive_plot
comparison_plots(1)
```

```
Frequency 374.0 GHz
################## Flux of Point Sources ##################
Sim     ngCASA   CASA
1.500   1.4935   1.4909
1.760   1.7518   1.7458
2.000   1.9950   1.9913
2.000   2.0391   2.0546
1.456   1.4630   1.4686
1.888   1.8954   1.8425
############ Percentage Difference Flux to Sim ############
ngCASA   CASA
0.4325   0.6079
0.4672   0.8043
0.2492   0.4330
-1.9533   -2.7304
-0.4825   -0.8683
-0.3894   2.4083
```



Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/imaging/alma_mosaic_test.ipynb

8.5 Alma Mosaic Imaging Test

This walkthrough is designed to be run in a Jupyter notebook on Google Colaboratory. To open the notebook in colab, go here.

8.5.1 Installation

[1]:
```
import os
os.system("pip install cngi-prototype==0.0.91")
print('complete')
```

```
complete
```

8.5.2 Dataset

[2]:
```
!gdown -q --id 16E-YPVLlgfth-2YVOdL8Uk3h3QT-73ck
!unzip Antennae_North_cal_lsrk.vis.zarr.zip > /dev/null

!gdown -q --id 1C_0ZarxgMo8mEdyDqs7IXf4IPlB7yzs9
!unzip Antennae_North_cal_lsrk_cube.img.zarr.zip > /dev/null

#%matplotlib widget
```

8.5.3 Load Dataset

[3]:
```
import xarray as xr
from cngi.dio import read_vis
#%matplotlib widget

xr.set_options(display_style="html")

infile = "Antennae_North_cal_lsrk.vis.zarr"
mxds = read_vis(infile)
print(mxds.xds0)
```

```
overwrite_encoded_chunks True
<xarray.Dataset>
Dimensions:         (baseline: 86, chan: 166, pol: 1, pol_id: 1, spw_id: 1, time:␣
↪1355, uvw_index: 3)
Coordinates:
  * baseline        (baseline) int64 0 1 2 3 4 5 6 7 ... 78 79 80 81 82 83 84 85
  * chan            (chan) float64 3.449e+11 3.449e+11 ... 3.43e+11 3.43e+11
    chan_width      (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
    effective_bw    (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
  * pol             (pol) int32 9
  * pol_id          (pol_id) int32 0
    resolution      (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
  * spw_id          (spw_id) int32 0
  * time            (time) datetime64[ns] 2011-05-28T01:41:17.375999451 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
```

(continues on next page)

(continued from previous page)

```
    ANTENNA1         (baseline) int32 dask.array<chunksize=(86,), meta=np.ndarray>
    ANTENNA2         (baseline) int32 dask.array<chunksize=(86,), meta=np.ndarray>
    ARRAY_ID         (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    DATA             (time, baseline, chan, pol) complex128 dask.array<chunksize=(1355,
↪ 86, 10, 1), meta=np.ndarray>
    DATA_WEIGHT      (time, baseline, chan, pol) float64 dask.array<chunksize=(1355,␣
↪86, 10, 1), meta=np.ndarray>
    EXPOSURE         (time, baseline) float64 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    ...               ...
    OBSERVATION_ID   (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    PROCESSOR_ID     (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    SCAN_NUMBER      (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    STATE_ID         (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    TIME_CENTROID    (time, baseline) float64 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    UVW              (time, baseline, uvw_index) float64 dask.array<chunksize=(1355,␣
↪86, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           1
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:
    net_sideband:     1
    num_chan:         166
    num_corr:         1
    ref_frequency:    344871541760.4001
    total_bandwidth:  1864427170.9199219
```

8.5.4 Grid Parameters

[4]:
```
grid_parms = {}
grid_parms['chan_mode'] = 'cube'
grid_parms['image_size'] = [500,500]
grid_parms['cell_size'] = [0.13,0.13]
grid_parms['fft_padding'] = 1.0
grid_parms['phase_center'] = mxds.FIELD.PHASE_DIR[12,0,:].data.compute()
```

8.5.5 Direction Rotation

The UVW coordinates must be rotated and the visbility DATA must be phase rotated, relative to the mosaic phase center specified by `rotation_parms['image_phase_center']`.

direction_rotate documentation

[5]:
```
#Reload modules
%load_ext autoreload
%autoreload 2
```

[6]:
```
from ngcasa.imaging import direction_rotate
import numpy as np
import dask
from cngi.vis import apply_flags

mxds0 = apply_flags(mxds, 'xds0', flags='FLAG')

xr.set_options(display_style="html")

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 0

rotation_parms = {}
rotation_parms['new_phase_center'] = grid_parms['phase_center']
rotation_parms['common_tangent_reprojection'] = True
rotation_parms['single_precision'] = True


mxds1 = direction_rotate(mxds0, rotation_parms, sel_parms)
mxds1.xds0.data_groups
```

```
########################## Start direction_rotate ##########################
Setting data_group_in  to  {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw': 'UVW',
↪'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '1',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT'}
########################## Created graph for direction_rotate ##########################
```

[6]:
```
[{'0': {'data': 'DATA',
   'flag': 'FLAG',
   'id': '0',
   'uvw': 'UVW',
   'weight': 'DATA_WEIGHT'},
  '1': {'data': 'DATA_ROT',
   'flag': 'FLAG',
   'id': '1',
   'uvw': 'UVW_ROT',
   'weight': 'DATA_WEIGHT'}}]
```

[7]:
```
print(mxds.xds0)
```

```
<xarray.Dataset>
Dimensions:          (baseline: 86, chan: 166, pol: 1, pol_id: 1, spw_id: 1, time:␣
↪1355, uvw_index: 3)
Coordinates:
  * baseline         (baseline) int64 0 1 2 3 4 5 6 7 ... 78 79 80 81 82 83 84 85
  * chan             (chan) float64 3.449e+11 3.449e+11 ... 3.43e+11 3.43e+11
```

(continues on next page)

(continued from previous page)

```
    chan_width      (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
    effective_bw    (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
  * pol             (pol) int32 9
  * pol_id          (pol_id) int32 0
    resolution      (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
  * spw_id          (spw_id) int32 0
  * time            (time) datetime64[ns] 2011-05-28T01:41:17.375999451 ... 2...
Dimensions without coordinates: uvw_index
Data variables: (12/17)
    ANTENNA1        (baseline) int32 dask.array<chunksize=(86,), meta=np.ndarray>
    ANTENNA2        (baseline) int32 dask.array<chunksize=(86,), meta=np.ndarray>
    ARRAY_ID        (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    DATA            (time, baseline, chan, pol) complex128 dask.array<chunksize=(1355,
↪ 86, 10, 1), meta=np.ndarray>
    DATA_WEIGHT     (time, baseline, chan, pol) float64 dask.array<chunksize=(1355,␣
↪86, 10, 1), meta=np.ndarray>
    EXPOSURE        (time, baseline) float64 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    ...              ...
    OBSERVATION_ID  (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    PROCESSOR_ID    (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    SCAN_NUMBER     (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    STATE_ID        (time, baseline) int32 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    TIME_CENTROID   (time, baseline) float64 dask.array<chunksize=(1355, 86), meta=np.
↪ndarray>
    UVW             (time, baseline, uvw_index) float64 dask.array<chunksize=(1355,␣
↪86, 3), meta=np.ndarray>
Attributes: (12/13)
    bbc_no:           1
    corr_product:     [[0, 0]]
    data_groups:      [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:       0
    freq_group_name:
    if_conv_chain:    0
    ...               ...
    name:
    net_sideband:     1
    num_chan:         166
    num_corr:         1
    ref_frequency:    344871541760.4001
    total_bandwidth:  1864427170.9199219
```

8.5.6 Make Imaging Weights

make_imaging_weight documentation

[8]:
```
from ngcasa.imaging import make_imaging_weight
import matplotlib.pylab as plt

imaging_weights_parms = {}
imaging_weights_parms['weighting'] = 'natural'

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 1

mxds2 = make_imaging_weight(mxds1, imaging_weights_parms, grid_parms, sel_parms)
```

```
######################### Start make_imaging_weights ########################
Setting data_group_in  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '1', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '1',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'IMAGING_WEIGHT'}
Since weighting is natural input weight will be reused as imaging weight.
######################### Created graph for make_imaging_weight ######################
↪###
```

8.5.7 Make Gridding Convolution Functions

make_gridding_convolution_function

[9]:
```
from ngcasa.imaging import make_gridding_convolution_function
import numpy as np
import dask.array as da
from cngi.dio import write_image

gcf_parms = {}
gcf_parms['function'] = 'casa_airy'
gcf_parms['list_dish_diameters'] = np.array([10.7])
gcf_parms['list_blockage_diameters'] = np.array([0.75])

unique_ant_indx = mxds.ANTENNA.DISH_DIAMETER.values
unique_ant_indx[unique_ant_indx == 12.0] = 0

gcf_parms['unique_ant_indx'] = unique_ant_indx.astype(int)
gcf_parms['phase_center'] = grid_parms['phase_center']

sel_parms = {}
sel_parms['xds'] = 'xds0'
sel_parms['data_group_in_id'] = 1

gcf_xds = make_gridding_convolution_function(mxds2, gcf_parms, grid_parms, sel_parms)
write_image(gcf_xds,'mosaic_gcf.gcf.zarr')
gcf_xds = xr.open_zarr('mosaic_gcf.gcf.zarr')
```

```
######################### Start make_gridding_convolution_function ##################
↪######
Setting data_group_in  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '1', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default chan_tolerance_factor  to  0.005
Setting default oversampling  to  [10, 10]
Setting default max_support  to  [15, 15]
Setting default support_cut_level  to  0.025
Setting default a_chan_num_chunk  to  3
Setting default image_center  to  [250 250]
#########################  Created graph for make_gridding_convolution_function ######
↪###################
Time to store and execute graph  write_zarr 0.9743752479553223
```

8.5.8 Make Mosaic Primary Beam and Image

make_mosaic_pb

make_image_with_gcf

[10]:
```
from ngcasa.imaging import make_mosaic_pb

vis_sel_parms = {}
vis_sel_parms['xds'] = 'xds0'
vis_sel_parms['data_group_in_id'] = 1

img_sel_parms = {}
img_xds= xr.Dataset()

img_xds = make_mosaic_pb(mxds2,gcf_xds,img_xds,vis_sel_parms,img_sel_parms,grid_parms)

################################################

from ngcasa.imaging import make_image_with_gcf
from cngi.dio import write_image, read_image

vis_select_parms = {}
vis_select_parms['xds'] = 'xds0'
vis_select_parms['data_group_in_id'] = 1

img_select_parms = {}
img_select_parms['data_group_in_id'] = 0

norm_parms = {}
norm_parms['norm_type'] = 'flat_sky'
#norm_parms['norm_type'] = 'none'

img_xds = make_image_with_gcf(mxds2,gcf_xds, img_xds, grid_parms, norm_parms, vis_
↪select_parms, img_select_parms)

write_image(img_xds,'mosaic_img.img.zarr')
img_xds = read_image('mosaic_img.img.zarr')
```

```
######################### Start make_mosaic_pb ##########################
Setting default image_center  to  [250 250]
Setting data_group_in  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '1', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
```

(continues on next page)

(continued from previous page)

```
Setting default data_group_in  to  {'id': '0'}
Setting default data_group_out  to  {'id': '0', 'pb': 'PB', 'weight_pb': 'WEIGHT_PB',
↪'weight_pb_sum_weight': 'WEIGHT_PB_SUM_WEIGHT'}
#########################  Created graph for make_mosaic_pb ########################
######################### Start make_image_with_gcf #########################
Setting default image_center  to  [250 250]
Setting default single_precision  to  True
Setting default pb_limit  to  0.2
Setting data_group_in  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '1', 'uvw':
↪'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting default data_group_out  to  {'data': 'DATA_ROT', 'flag': 'FLAG', 'id': '2',
↪'uvw': 'UVW_ROT', 'weight': 'DATA_WEIGHT', 'imaging_weight': 'DATA_WEIGHT'}
Setting data_group_in  to  {'id': '0', 'pb': 'PB', 'weight_pb': 'WEIGHT_PB', 'weight_
↪pb_sum_weight': 'WEIGHT_PB_SUM_WEIGHT'}
Setting default data_group_out  to  {'pb': 'PB', 'weight_pb': 'WEIGHT_PB', 'id': '1',
↪'weight_pb_sum_weight': 'WEIGHT_PB_SUM_WEIGHT', 'sum_weight': 'SUM_WEIGHT', 'image':
↪ 'IMAGE'}
grid sizes 500 500
#########################  Created graph for make_mosaic_with_gcf ###################
↪#####
```

```
/usr/local/lib/python3.7/dist-packages/ngcasa/imaging/_imaging_utils/_normalize.py:51:
↪ RuntimeWarning: invalid value encountered in true_divide
  normalized_image = (image / sum_weights_copy) / (oversampling_correcting_func[:,:,
↪None,None]*normalizing_image)
```

```
Time to store and execute graph  write_zarr 40.8763325214386
```

8.5.9 Compare CASA and ngCASA Primary Beams

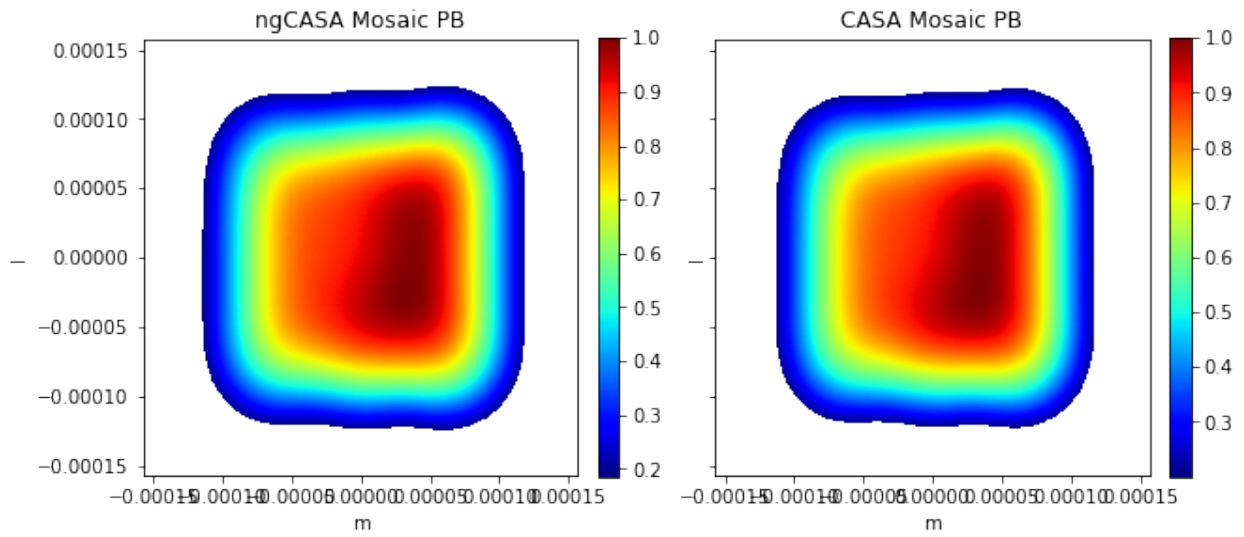
[11]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive
import scipy
from scipy.signal import decimate
from cngi.image import implot
import xarray as xr

img_xds = xr.open_zarr('mosaic_img.img.zarr').isel(time=0,pol=0)
casa_img_xds = xr.open_zarr('Antennae_North_cal_lsrk_cube.img.zarr').isel(time=0,
↪pol=0)
pb_limit = 0.2
extent = extent=(np.min(casa_img_xds.m),np.max(casa_img_xds.m),np.min(casa_img_xds.l),
↪np.max(casa_img_xds.l))

#print(casa_img_xds)

def comparison_plots(chan):
    plt.close('all')
    print('Frequency',img_xds.chan[chan].values/10**9, 'GHz')
    mosaic_pb = img_xds.PB.isel(chan=chan)
    mosaic_pb = mosaic_pb.where(mosaic_pb > pb_limit,other=np.nan)
    mosaic_pb = mosaic_pb/np.max(mosaic_pb)

    casa_mosaic_pb = casa_img_xds.PB.isel(chan=chan)
    casa_mosaic_pb = casa_mosaic_pb.where(casa_mosaic_pb > pb_limit,other=np.nan)
```

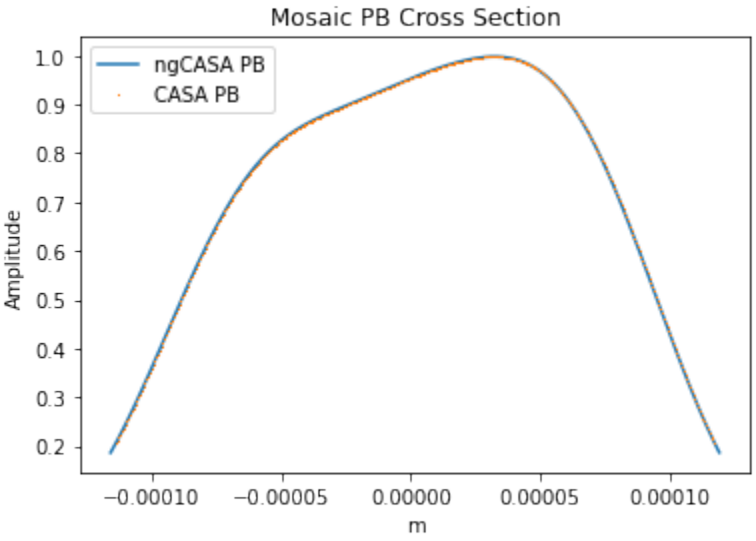
(continues on next page)

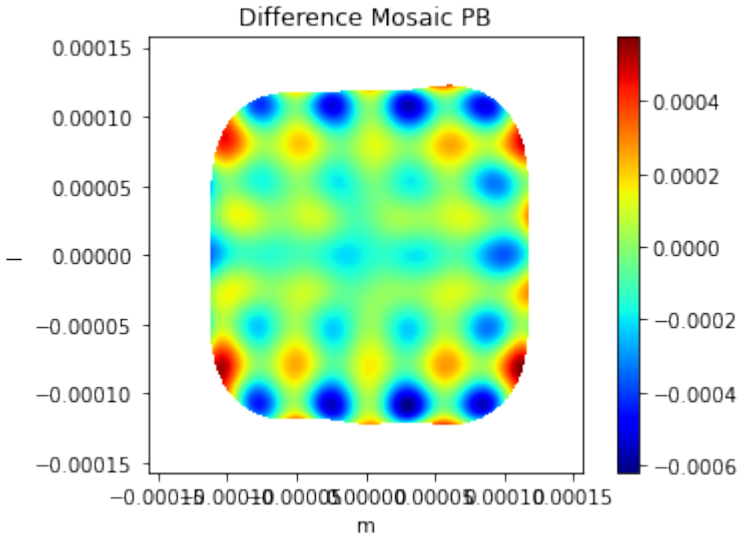
(continued from previous page)

```
    fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
    im0 = ax0[0].imshow(mosaic_pb,cmap='jet',extent=extent)
    im1 = ax0[1].imshow(casa_mosaic_pb,cmap='jet',extent=extent)
    ax0[0].title.set_text('ngCASA Mosaic PB')
    ax0[1].title.set_text('CASA Mosaic PB')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'), ax0[1].
↪set_ylabel('l')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)

    plt.figure()
    plt.plot(casa_mosaic_pb.m,mosaic_pb.isel(l=300),label='ngCASA PB')
    plt.plot(casa_mosaic_pb.m,casa_mosaic_pb.isel(l=300),'*',label='CASA PB',
↪markersize=0.5)
    plt.legend()
    plt.xlabel('m')
    plt.ylabel('Amplitude')
    plt.title('Mosaic PB Cross Section')

    diff_image = mosaic_pb - casa_mosaic_pb
    plt.figure()
    plt.imshow(mosaic_pb - casa_mosaic_pb,extent=extent,cmap='jet')
    plt.xlabel('m'), plt.ylabel('l')
    plt.colorbar()
    plt.title('Difference Mosaic PB')
    plt.show()

#interactive_plot = interactive(comparison_plots, chan=(0, len(img_xds.chan)-1))
#output = interactive_plot.children[-1]
#output.layout.auto_scroll_threshold = 9999;
#interactive_plot
comparison_plots(82)
```

```
Frequency 343.9505596639216 GHz
```







8.5.10 Compare CASA and ngCASA Sky Images

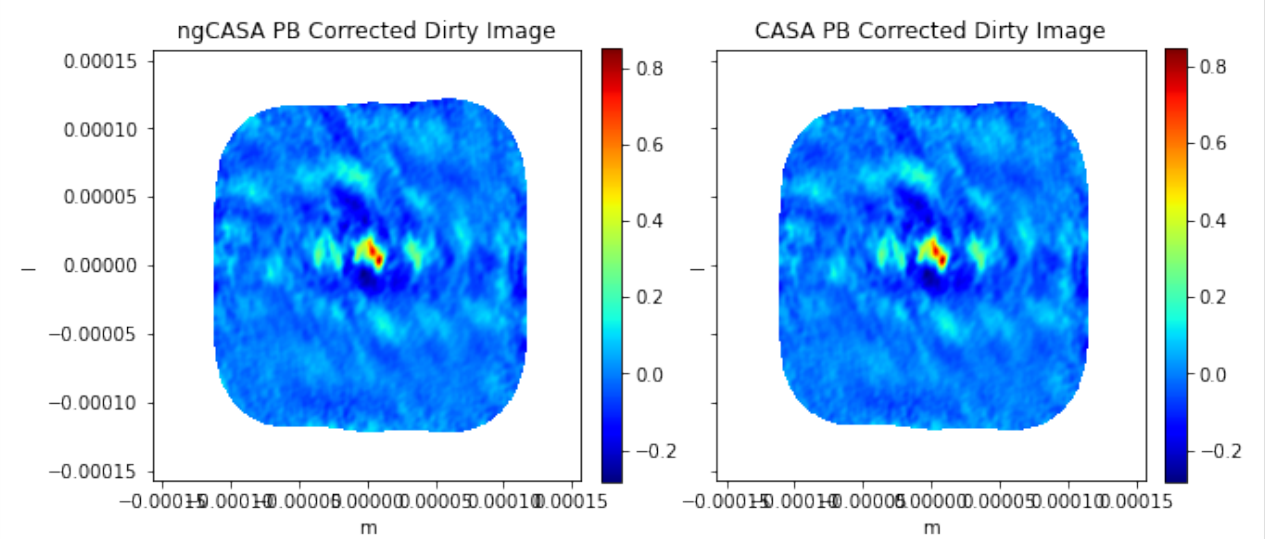
[12]:
```
import matplotlib.pylab as plt
import numpy as np
from ipywidgets import interactive
import scipy
from scipy.signal import decimate
import xarray as xr

img_xds = xr.open_zarr('mosaic_img.img.zarr',chunks={'l':grid_parms['image_size'][0],
↪'m':grid_parms['image_size'][1]}).isel(time=0,pol=0)
casa_img_xds = xr.open_zarr('Antennae_North_cal_lsrk_cube.img.zarr',chunks={'l':grid_
↪parms['image_size'][0],'m':grid_parms['image_size'][1]}).isel(time=0,pol=0)
pb_limit = 0.22
extent = extent=(np.min(casa_img_xds.m),np.max(casa_img_xds.m),np.min(casa_img_xds.l),
↪np.max(casa_img_xds.l))
```

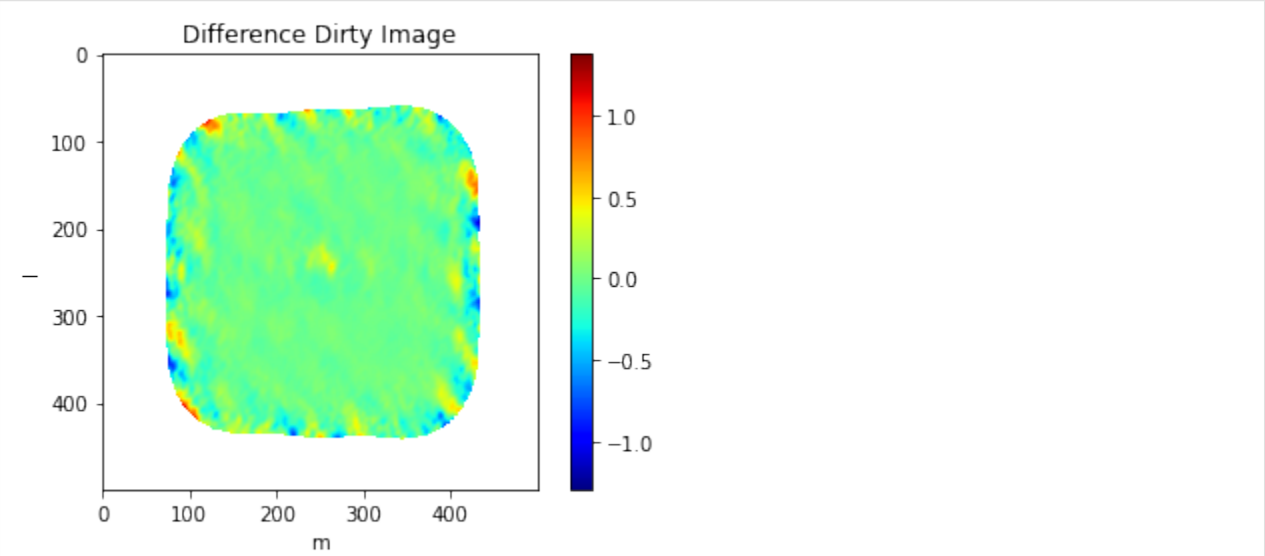
(continues on next page)

(continued from previous page)

```
ngcasa_image_name = 'IMAGE'
casa_image_name = 'IMAGE_PBCOR'
#casa_image_name = 'IMAGE'

def comparison_plots(chan):
    print('Frequency',img_xds.chan.isel(chan=chan).values/10**9, 'GHz')
    mosaic_pb = img_xds.PB.isel(chan=chan)
    casa_mosaic_pb = casa_img_xds.PB.isel(chan=chan)

    mosaic_img = img_xds[ngcasa_image_name].isel(chan=chan)
    mosaic_img = mosaic_img.where(mosaic_pb > pb_limit,other=np.nan)

    casa_mosaic_img = casa_img_xds[casa_image_name].isel(chan=chan)
    casa_mosaic_img = casa_mosaic_img.where(casa_mosaic_pb > pb_limit,other=np.nan)

    fig0, ax0 = plt.subplots(1, 2, sharey=True,figsize=(10, 5))
    im0 = ax0[0].imshow(mosaic_img,cmap='jet',extent=extent)
    im1 = ax0[1].imshow(casa_mosaic_img,cmap='jet',extent=extent)
    ax0[0].title.set_text('ngCASA PB Corrected Dirty Image')
    ax0[1].title.set_text('CASA PB Corrected Dirty Image')
    ax0[0].set_xlabel('m'), ax0[1].set_xlabel('m'), ax0[0].set_ylabel('l'), ax0[1].
↪set_ylabel('l')
    fig0.colorbar(im0, ax=ax0[0], fraction=0.046, pad=0.04)
    fig0.colorbar(im1, ax=ax0[1], fraction=0.046, pad=0.04)

    plt.figure()
    plt.imshow((100*(mosaic_img.data - casa_mosaic_img.data)/np.nanmax(casa_mosaic_
↪img.data)),cmap='jet')
    plt.title('Difference Dirty Image')
    plt.xlabel('m'), plt.ylabel('l')
    plt.colorbar()
    plt.show()

#interactive_plot = interactive(comparison_plots, chan=(0, len(img_xds.chan)-1))
#output = interactive_plot.children[-1]
#output.layout.auto_scroll_threshold = 9999;
#interactive_plot
comparison_plots(82)
```

```
Frequency 343.9505596639216 GHz
```





[13]:
```
from cngi.image.moments import moments

chan_slice = slice(0,139)
img_xds.attrs['rest_frequency'] = casa_img_xds.attrs['rest_frequency']

pb_limit = 0.25
img_xds_pb_cut = img_xds.copy(deep=True)
img_xds_pb_cut['IMAGE'] = img_xds['IMAGE'].where(img_xds['PB'].data > pb_limit,
↪other=np.nan)
img_xds_mom8 = moments(img_xds_pb_cut.isel(chan=chan_slice), moment=8, axis='chan')
img_mom8 = img_xds_mom8.MOMENTS_MAXIMUM

casa_img_xds_pb_cut = casa_img_xds.copy(deep=True)
casa_img_xds_pb_cut['IMAGE'] = casa_img_xds['IMAGE_PBCOR'].where(img_xds['PB'].data >␣
↪pb_limit,other=np.nan)
casa_img_xds_mom8 = moments(casa_img_xds_pb_cut.isel(chan=chan_slice), moment=8, axis=
↪'chan')
casa_img_mom8 = casa_img_xds_mom8.MOMENTS_MAXIMUM
```

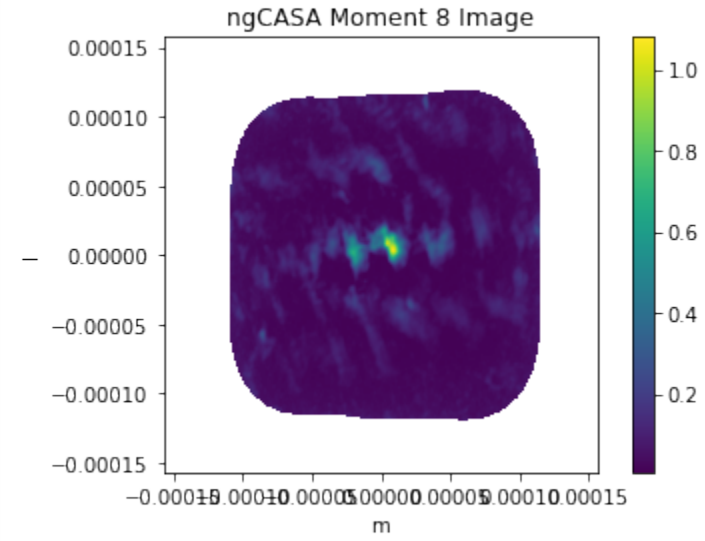
(continues on next page)

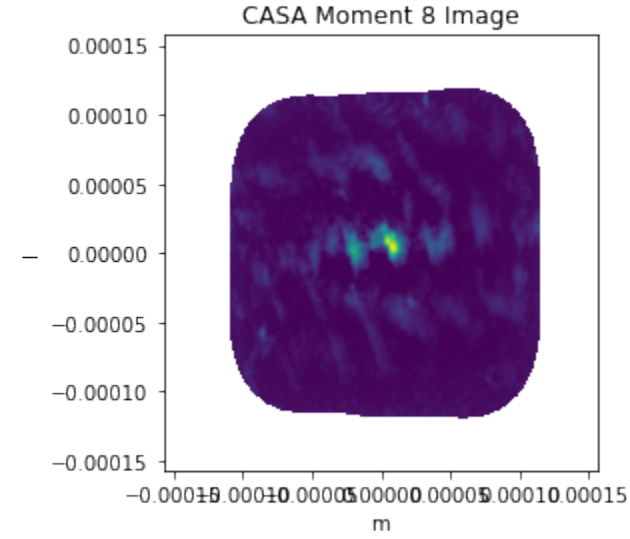
(continued from previous page)

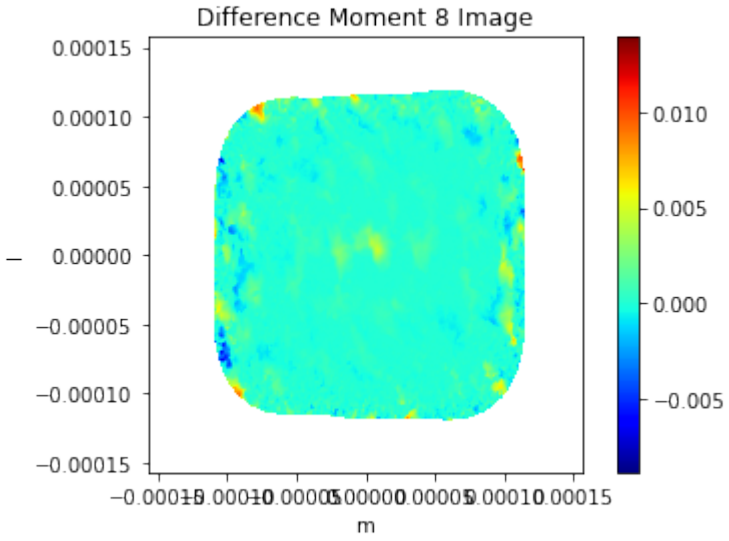
```
plt.figure()
plt.imshow(img_mom8,extent=extent)
plt.title('ngCASA Moment 8 Image')
plt.xlabel('m'), plt.ylabel('l')
plt.colorbar()

plt.figure()
plt.imshow(casa_img_mom8,extent=extent)
plt.title('CASA Moment 8 Image')
plt.xlabel('m'), plt.ylabel('l')

plt.figure()
plt.imshow((img_mom8 - casa_img_mom8)/np.nanmax(casa_img_mom8),cmap='jet',
↪extent=extent)
plt.title('Difference Moment 8 Image')
plt.xlabel('m'), plt.ylabel('l')
plt.colorbar()
```

```
/usr/local/lib/python3.7/dist-packages/dask/utils.py:35: RuntimeWarning: All-NaN␣
↪slice encountered
  return func(*args, **kwargs)
/usr/local/lib/python3.7/dist-packages/toolz/functoolz.py:488: RuntimeWarning: All-
↪NaN slice encountered
  ret = f(ret)
/usr/local/lib/python3.7/dist-packages/dask/utils.py:35: RuntimeWarning: All-NaN␣
↪slice encountered
  return func(*args, **kwargs)
/usr/local/lib/python3.7/dist-packages/toolz/functoolz.py:488: RuntimeWarning: All-
↪NaN slice encountered
  ret = f(ret)
/usr/local/lib/python3.7/dist-packages/dask/utils.py:35: RuntimeWarning: All-NaN␣
↪slice encountered
  return func(*args, **kwargs)
/usr/local/lib/python3.7/dist-packages/toolz/functoolz.py:488: RuntimeWarning: All-
↪NaN slice encountered
  ret = f(ret)
/usr/local/lib/python3.7/dist-packages/dask/utils.py:35: RuntimeWarning: All-NaN␣
↪slice encountered
  return func(*args, **kwargs)
/usr/local/lib/python3.7/dist-packages/toolz/functoolz.py:488: RuntimeWarning: All-
↪NaN slice encountered
  ret = f(ret)
```

[13]:
```
<matplotlib.colorbar.Colorbar at 0x7f0d0d3cc890>
```

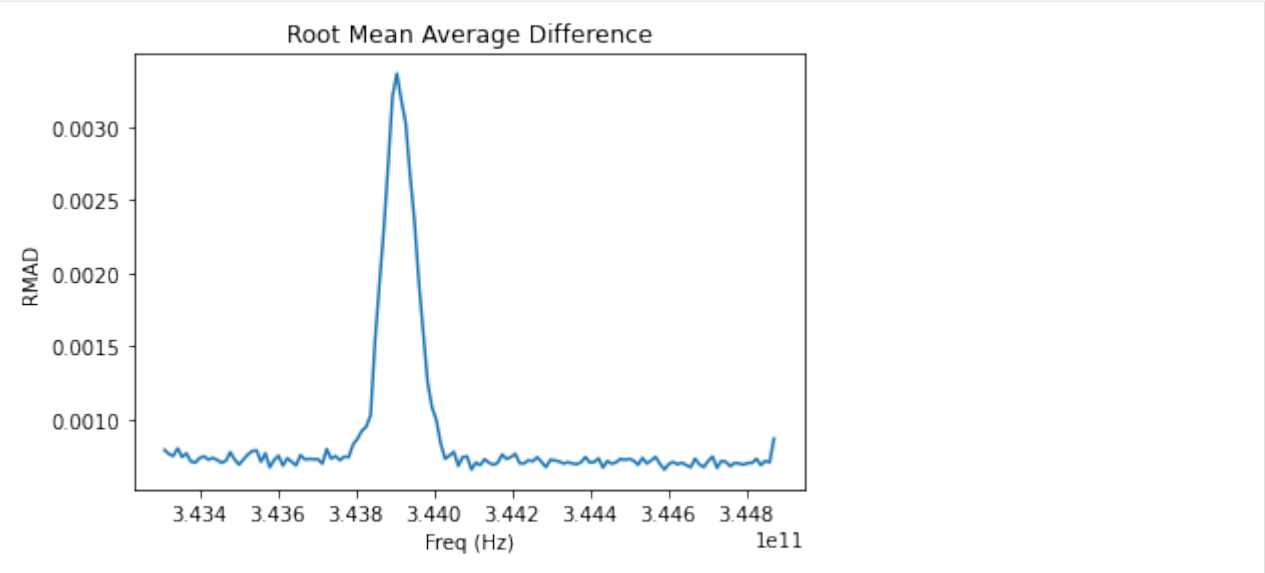

ngCASA Moment 8 Image
0.00015
0.00010
0.00005
0.00000
−0.00005
−0.00010
−0.00015
m
1.0
0.8
0.6
0.4
0.2


CASA Moment 8 Image
0.00015
0.00010
0.00005
0.00000
−0.00005
−0.00010
−0.00015
m



[14]:
```
from cngi.image.moments import moments


pb_limit = 0.2
img_xds_pb_cut = img_xds.copy(deep=True)
ngcasa_pbcor = img_xds['IMAGE'].where(img_xds['PB'].data > pb_limit,other=np.nan)

casa_img_xds_pb_cut = casa_img_xds.copy(deep=True)
casa_pbcor = casa_img_xds['IMAGE_PBCOR'].where(img_xds['PB'].data > pb_limit,other=np.
↪nan)

rmad_per_channel = np.nanmean(np.abs(ngcasa_pbcor.data-casa_pbcor.data),axis=(0,1))

print(ngcasa_pbcor)
plt.figure()
plt.plot(ngcasa_pbcor.chan[:140],rmad_per_channel[:140])
plt.title('Root Mean Average Difference')
plt.ylabel('RMAD')
plt.xlabel('Freq (Hz)')
plt.show()
```

```
<xarray.DataArray 'IMAGE' (l: 500, m: 500, chan: 166)>
dask.array<where, shape=(500, 500, 166), dtype=float64, chunksize=(500, 500, 10),␣
↪chunktype=numpy.ndarray>
Coordinates:
  * chan             (chan) float64 3.449e+11 3.449e+11 ... 3.43e+11 3.43e+11
    chan_width       (chan) float64 dask.array<chunksize=(10,), meta=np.ndarray>
    declination      (l, m) float64 dask.array<chunksize=(500, 500), meta=np.ndarray>
  * l                (l) float64 0.0001576 0.0001569 ... -0.0001563 -0.0001569
  * m                (m) float64 -0.0001576 -0.0001569 ... 0.0001563 0.0001569
    pol              int32 9
    right_ascension  (l, m) float64 dask.array<chunksize=(500, 500), meta=np.ndarray>
    time             datetime64[ns] 2011-06-04T07:26:13.328236156
```



Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/benchmarking.ipynb

CHAPTER

NINE

BENCHMARKS

A demonstration of parallel processing performance of this CNGI prototype against the current released version of CASA using a selection of ALMA datasets representing different computationally demanding configurations and a subset of the VLA CHILES Survey data.

9.1 Methodology

Measurement of runtime performance for the component typically dominating compute cost for existing and future workflows – the major cycle – was made against the reference implementation of CASA 6.2. Relevant calls of the CASA task `tclean` were isolated for direct comparison with the latest version of the `cngi-prototype` implementation of the mosaic and standard gridders.

The steps of the workflow used to prepare data for testing were:

1. Download archive data from ALMA Archive
2. Restore calibrated MeasurementSet using scriptForPI.py with compatible version of CASA
3. Split off science targets and representative spectral window into a single MeasurementSet
4. Convert calibrated MeasurementSet into zarr format using `cngi.conversion.convert_ms`

This allowed for generation of image data from visibilities for comparison. Tests were run in two different environments:

1. On premises using the same high performance computing (HPC) cluster environment used for offline processing of data from North American ALMA operations.
2. Using commercial cloud resources furnished by Amazon Web Services (AWS).

9.2 Dataset Selection

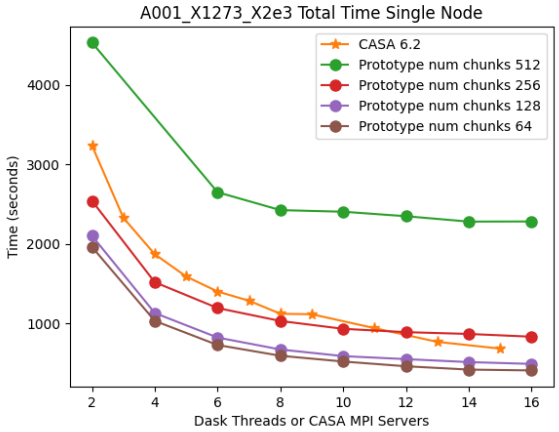
Observations were chosen for their source/observation properties, data volume, and usage mode diversity, particularly the relatively large number of spectral channels, pointings, or executions. Two were observed by the ALMA Compact Array (ACA) of 7m antennas, and two were observed by the main array of 12m antennas.

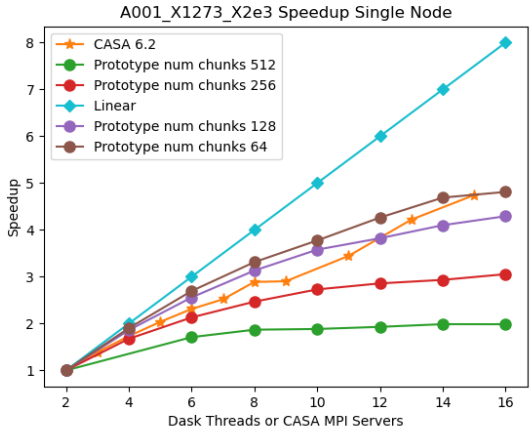
The datasets from each project code and Member Object Unit Set (MOUS) were processed following publicly documented ALMA archival reprocessing workflows, and come from public observations used used by other teams in previous benchmarking and profiling efforts.

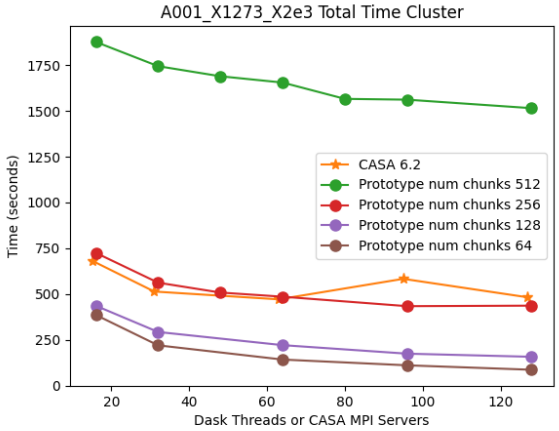
9.2.1 2017.1.00271.S

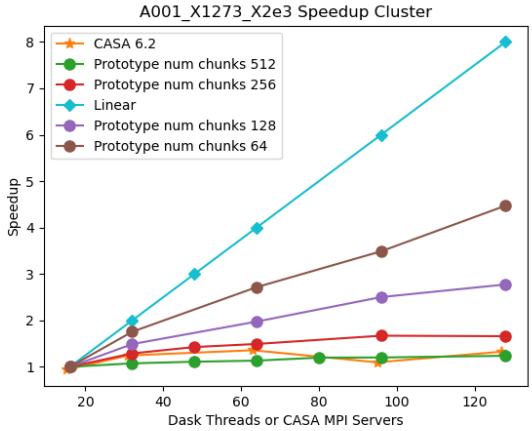
Compact array observations over many (nine) execution blocks using the mosaic gridder.

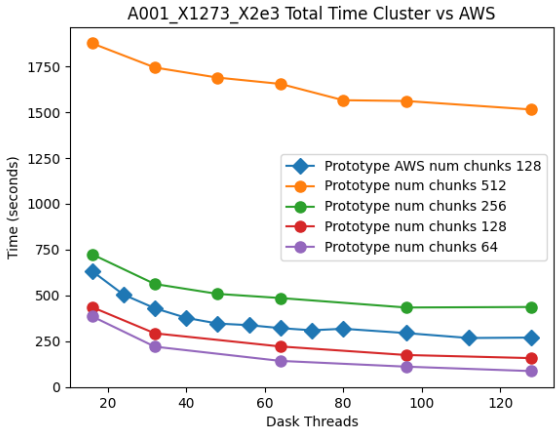
- MOUS uid://A001/X1273/X2e3
- Measurement Set Rows: 100284
- CNGI Shape (time, baseline, chan, pol): (2564, 53, 2048, 2)
- Image size (x,y,chan,pol): (400, 500, 2048, 2)
- Data Volume (vis.zarr and img.zarr): 30 GB

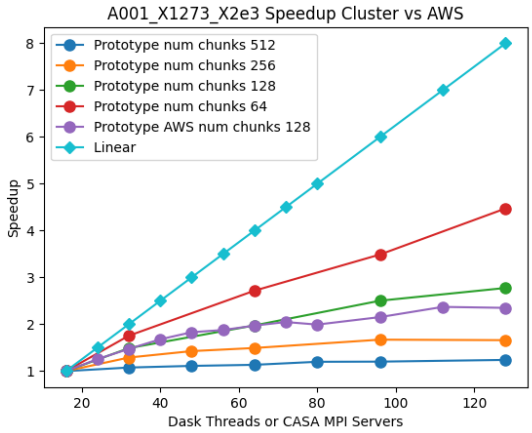

A001_X1273_X2e3 Total Time Single Node
CASA 6.2
Prototype num chunks 512
Prototype num chunks 256
Prototype num chunks 128
Prototype num chunks 64
Time (seconds)
Dask Threads or CASA MPI Servers


A001_X1273_X2e3 Speedup Single Node
CASA 6.2
Prototype num chunks 512
Prototype num chunks 256
Linear
Prototype num chunks 128
Prototype num chunks 64
Speedup
Dask Threads or CASA MPI Servers


A001_X1273_X2e3 Total Time Cluster
CASA 6.2
Prototype num chunks 512
Prototype num chunks 256
Prototype num chunks 128
Prototype num chunks 64
Time (seconds)
Dask Threads or CASA MPI Servers


A001_X1273_X2e3 Speedup Cluster
CASA 6.2
Prototype num chunks 512
Prototype num chunks 256
Linear
Prototype num chunks 128
Prototype num chunks 64
Speedup
Dask Threads or CASA MPI Servers

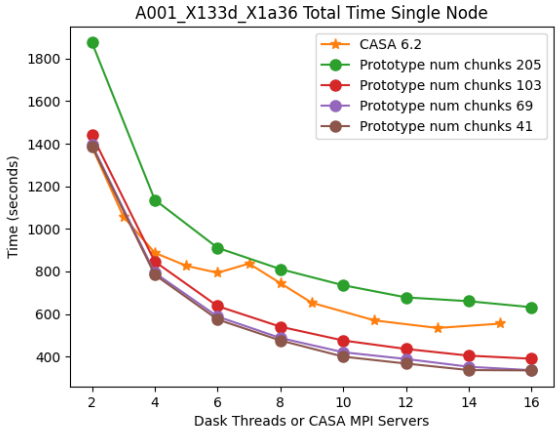

A001_X1273_X2e3 Total Time Cluster vs AWS
Prototype AWS num chunks 128
Prototype num chunks 512
Prototype num chunks 256
Prototype num chunks 128
Prototype num chunks 64
Time (seconds)
Dask Threads

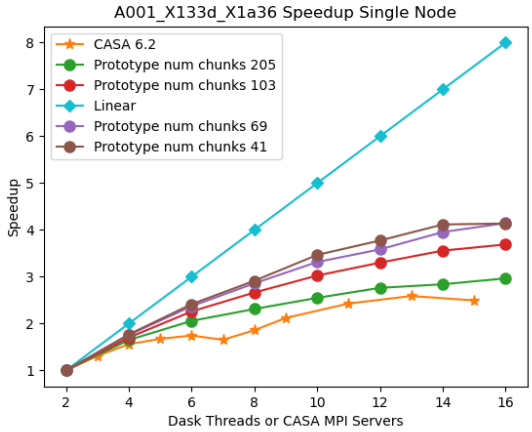

A001_X1273_X2e3 Speedup Cluster vs AWS
Prototype num chunks 512
Prototype num chunks 256
Prototype num chunks 128
Prototype num chunks 64
Prototype AWS num chunks 128
Linear
Speedup
Dask Threads or CASA MPI Servers

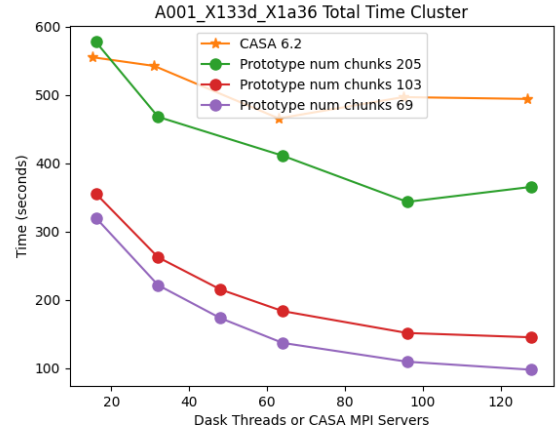
9.2.2 2018.1.01091.S

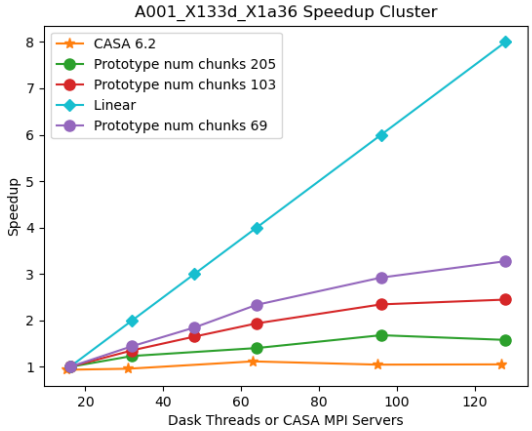
Compact array observations with many (141) pointings using the mosaic gridder.

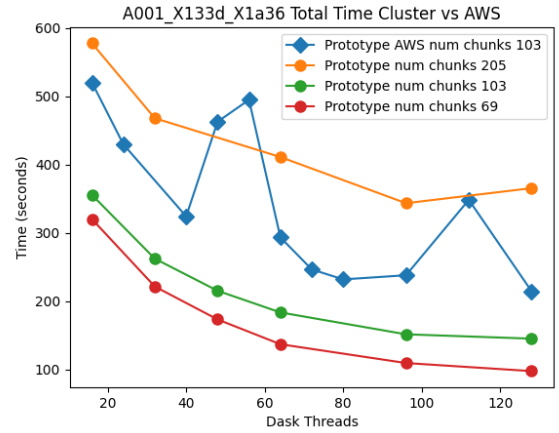
- MOUS uid://A001/X133d/X1a36
- Measurement Set Rows: 31020
- CNGI Shape (time, baseline, chan, pol): (564, 55, 1025, 2)
- Image size (x,y,chan,pol): (600, 1000, 1025, 2)
- Data Volume (vis.zarr and img.zarr): 31 GB

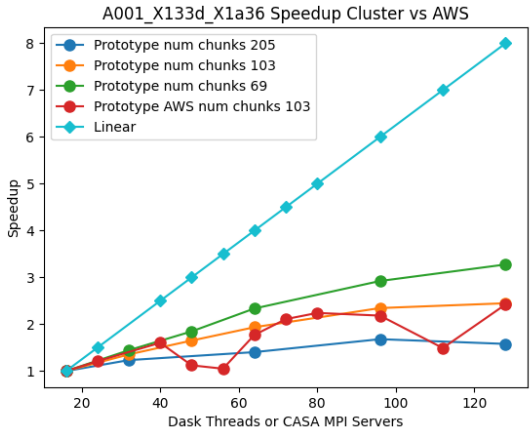

A001_X133d_X1a36 Total Time Single Node
CASA 6.2
Prototype num chunks 205
Prototype num chunks 103
Prototype num chunks 69
Prototype num chunks 41
Time (seconds)
Dask Threads or CASA MPI Servers


A001_X133d_X1a36 Speedup Single Node
CASA 6.2
Prototype num chunks 205
Prototype num chunks 103
Linear
Prototype num chunks 69
Prototype num chunks 41
Speedup
Dask Threads or CASA MPI Servers


A001_X133d_X1a36 Total Time Cluster
CASA 6.2
Prototype num chunks 205
Prototype num chunks 103
Prototype num chunks 69
Time (seconds)
Dask Threads or CASA MPI Servers


A001_X133d_X1a36 Speedup Cluster
CASA 6.2
Prototype num chunks 205
Prototype num chunks 103
Linear
Prototype num chunks 69
Speedup
Dask Threads or CASA MPI Servers

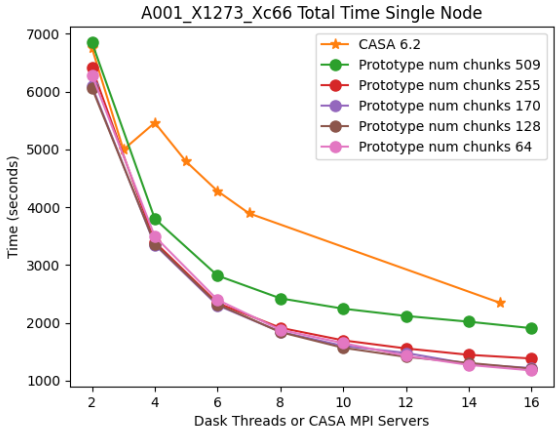

A001_X133d_X1a36 Total Time Cluster vs AWS
Prototype AWS num chunks 103
Prototype num chunks 205
Prototype num chunks 103
Prototype num chunks 69
Time (seconds)
Dask Threads

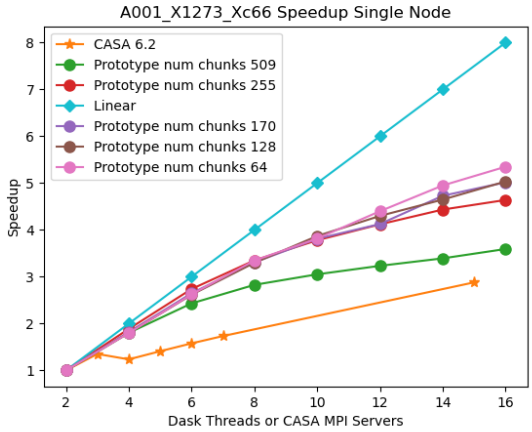

A001_X133d_X1a36 Speedup Cluster vs AWS
Prototype num chunks 205
Prototype num chunks 103
Prototype num chunks 69
Prototype AWS num chunks 103
Linear
Speedup
Dask Threads or CASA MPI Servers

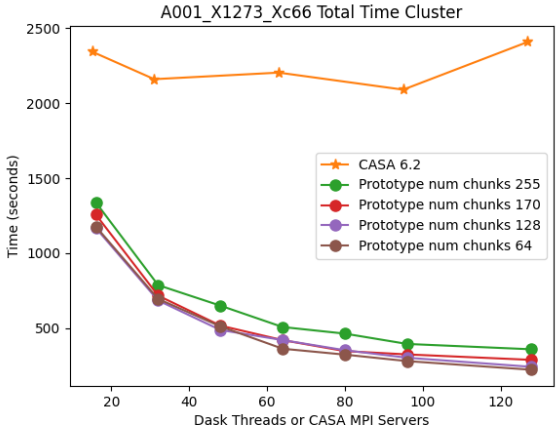
9.2.3 2017.1.00717.S

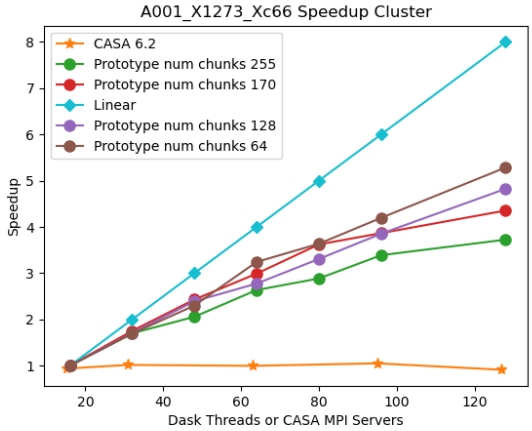
Main array observations with many spectral channels and visibilities using the standard gridder.

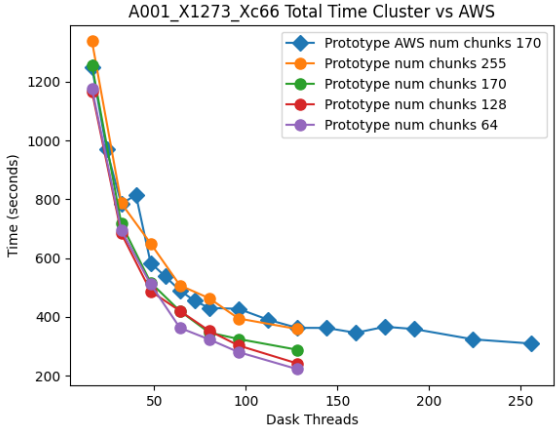
- MOUS uid://A001/X1273/Xc66
- Measurement Set Rows: 315831
- CNGI Shape (time, baseline, chan, pol): (455, 745, 7635, 2)
- Image size (x,y,chan,pol): (600, 600, 7635, 2)
- Data Volume (vis.zarr and img.zarr): 248 GB

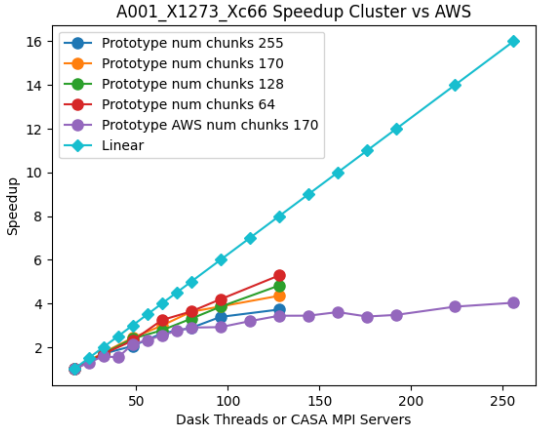

A001_X1273_Xc66 Total Time Single Node
Time (seconds)
Dask Threads or CASA MPI Servers
CASA 6.2
Prototype num chunks 509
Prototype num chunks 255
Prototype num chunks 170
Prototype num chunks 128
Prototype num chunks 64


A001_X1273_Xc66 Speedup Single Node
Speedup
Dask Threads or CASA MPI Servers
CASA 6.2
Prototype num chunks 509
Prototype num chunks 255
Linear
Prototype num chunks 170
Prototype num chunks 128
Prototype num chunks 64


A001_X1273_Xc66 Total Time Cluster
Time (seconds)
Dask Threads or CASA MPI Servers
CASA 6.2
Prototype num chunks 255
Prototype num chunks 170
Prototype num chunks 128
Prototype num chunks 64


A001_X1273_Xc66 Speedup Cluster
Speedup
Dask Threads or CASA MPI Servers
CASA 6.2
Prototype num chunks 255
Prototype num chunks 170
Linear
Prototype num chunks 128
Prototype num chunks 64

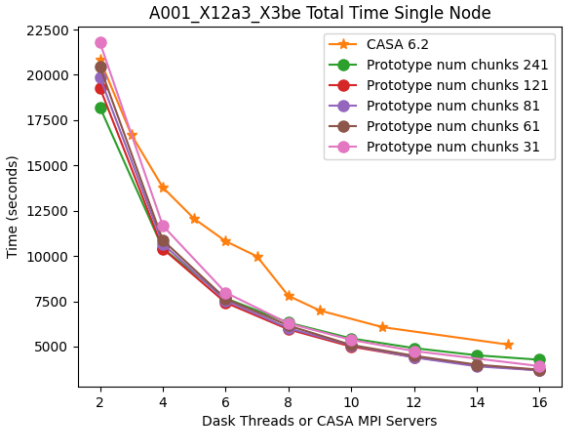

A001_X1273_Xc66 Total Time Cluster vs AWS
Time (seconds)
Dask Threads
Prototype AWS num chunks 170
Prototype num chunks 255
Prototype num chunks 170
Prototype num chunks 128
Prototype num chunks 64

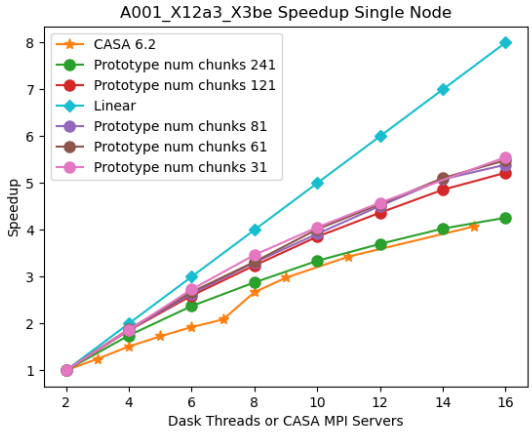

A001_X1273_Xc66 Speedup Cluster vs AWS
Speedup
Dask Threads or CASA MPI Servers
Prototype num chunks 255
Prototype num chunks 170
Prototype num chunks 128
Prototype num chunks 64
Prototype AWS num chunks 170
Linear

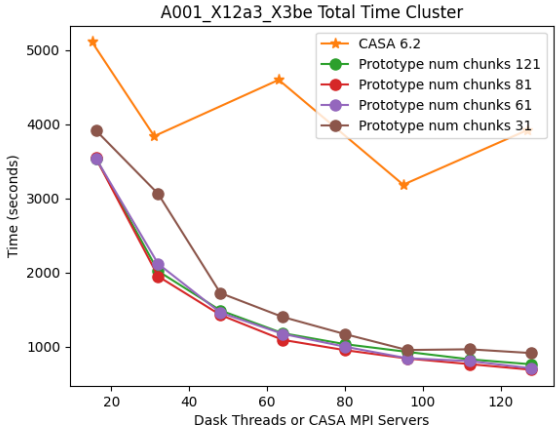
9.2.4 2017.1.00983.S

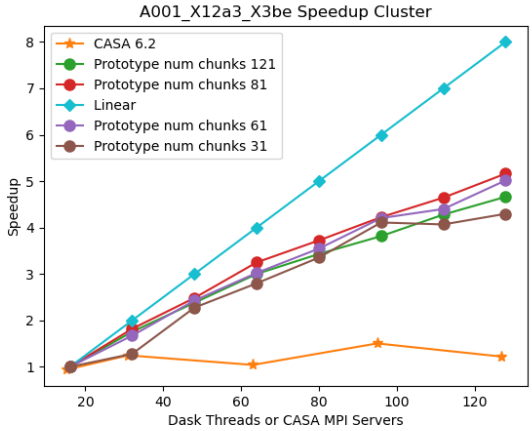
Main array observations with many spectral channels and visibilities using the mosaic gridder.

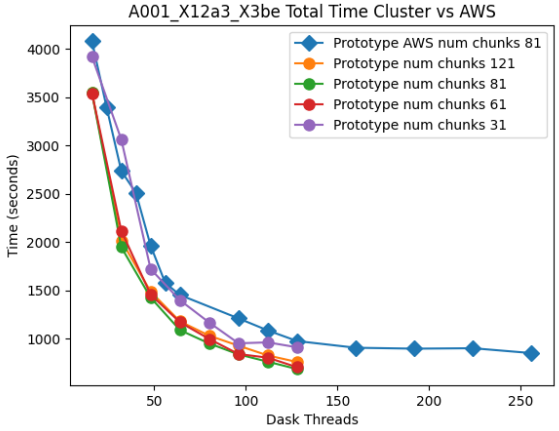
- MOUS uid://A001/X12a3/X3be
- Measurement Set Rows: 646418
- CNGI Shape (time, baseline, chan, pol): (729, 1159, 3853, 2)
- Image size (x,y,chan,pol): (1000, 1000, 3853, 2)
- Data Volume (vis.zarr and img.zarr): 304 GB

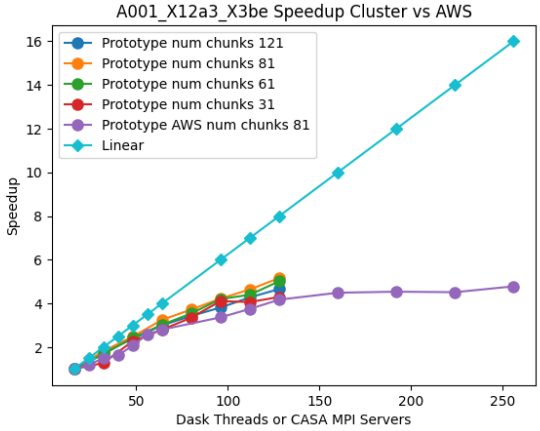

A001_X12a3_X3be Total Time Single Node
CASA 6.2
Prototype num chunks 241
Prototype num chunks 121
Prototype num chunks 81
Prototype num chunks 61
Prototype num chunks 31
Time (seconds)
Dask Threads or CASA MPI Servers


A001_X12a3_X3be Speedup Single Node
CASA 6.2
Prototype num chunks 241
Prototype num chunks 121
Linear
Prototype num chunks 81
Prototype num chunks 61
Prototype num chunks 31
Speedup
Dask Threads or CASA MPI Servers


A001_X12a3_X3be Total Time Cluster
CASA 6.2
Prototype num chunks 121
Prototype num chunks 81
Prototype num chunks 61
Prototype num chunks 31
Time (seconds)
Dask Threads or CASA MPI Servers


A001_X12a3_X3be Speedup Cluster
CASA 6.2
Prototype num chunks 121
Prototype num chunks 81
Linear
Prototype num chunks 61
Prototype num chunks 31
Speedup
Dask Threads or CASA MPI Servers


A001_X12a3_X3be Total Time Cluster vs AWS
Prototype AWS num chunks 81
Prototype num chunks 121
Prototype num chunks 81
Prototype num chunks 61
Prototype num chunks 31
Time (seconds)
Dask Threads


A001_X12a3_X3be Speedup Cluster vs AWS
Prototype num chunks 121
Prototype num chunks 81
Prototype num chunks 61
Prototype num chunks 31
Prototype AWS num chunks 81
Linear
Speedup
Dask Threads or CASA MPI Servers

9.3 Comparison of Runtimes

Single Machine

The total runtime of the prototype has comparable performance to the CASA 6.2 reference implementations for all datasets. There does not appear to be a performance penalty associated with adopting a pure Python framework compared to the compiled C++/Fortran reference implementation. This is likely due largely to the prototype's reliance on the `numba` Just-In-Time (JIT) transpiler, and the C foreign function interface relied on by third-party framework packages including `numpy` and `scipy`.

The Fortran gridding code in CASA is slightly more efficient than the JIT-decorated Python code in the prototype. However, the test implementation more efficiently handles chunked data and does not have intermediate steps where data is written to disk, whereas CASA generates TempLattice files to store intermediate files.

Multi-Node

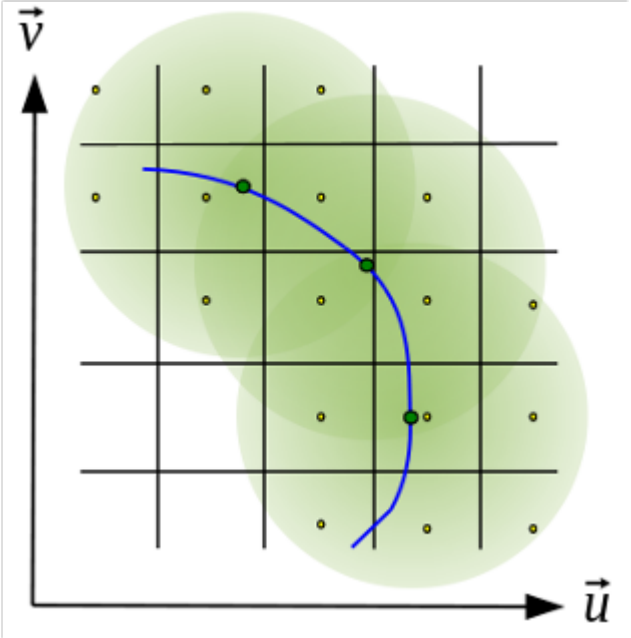
The total runtime of the prototype mosaic and standard gridders was less than the 6.2 reference implementations except for a couple of suboptimal chunking sizes. For the two larger datasets 2017.1.00717.S and 2017.1.00983.S, a significant speedup of up to six times and ten times, respectively, were achieved. Furthermore, the performance of CASA 6.2 stagnates after a single node because the CASACORE table system can not write the output images to disk in parallel.

Chunking

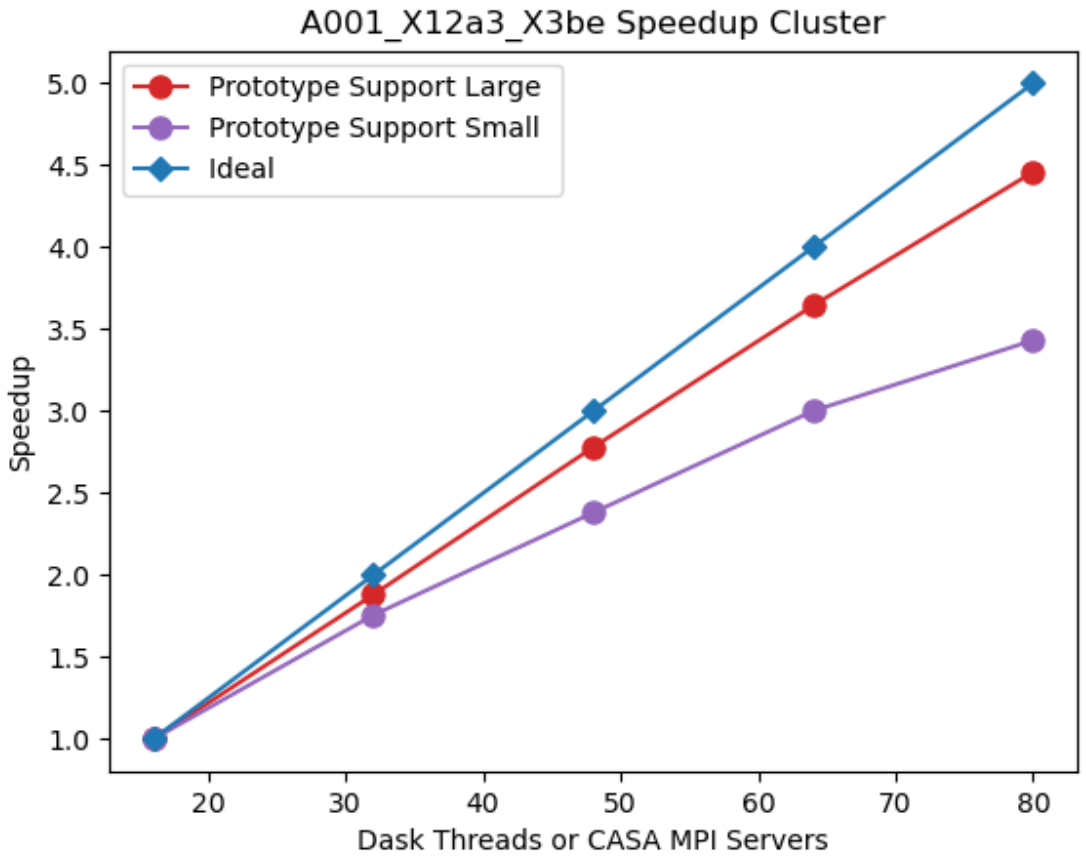
The chunking of the data determines the number of nodes in Dask graph. As the number of chunks increases, so does scheduler overhead. However, there might not be enough work to occupy all the nodes if there are too few chunks. If data associated with a chunk is too large to fit into memory, the Dask scheduler will spill intermediate data products to disk.

Linear Speedup

Neither CASA nor the prototype achieves a linear speedup due to scheduler overhead, disk io, internode communication, etc. The most computationally expensive step is the gridding of the convolved visibilities onto the uv grid:



Since the gridding step can be done in parallel, increasing gridding convolution support will increase the part of the work that can be done in parallel. In the figure below, the support size of the gridding convolution kernel is increased from 17x17 to 51x51, consequently, a more linear speedup is achieved.



9.4 CHILES Benchmark

The CHILES experiment is an HI large extragalactic survey with over 1000 hours of observing time on the VLA. We choose a subset of the survey data:

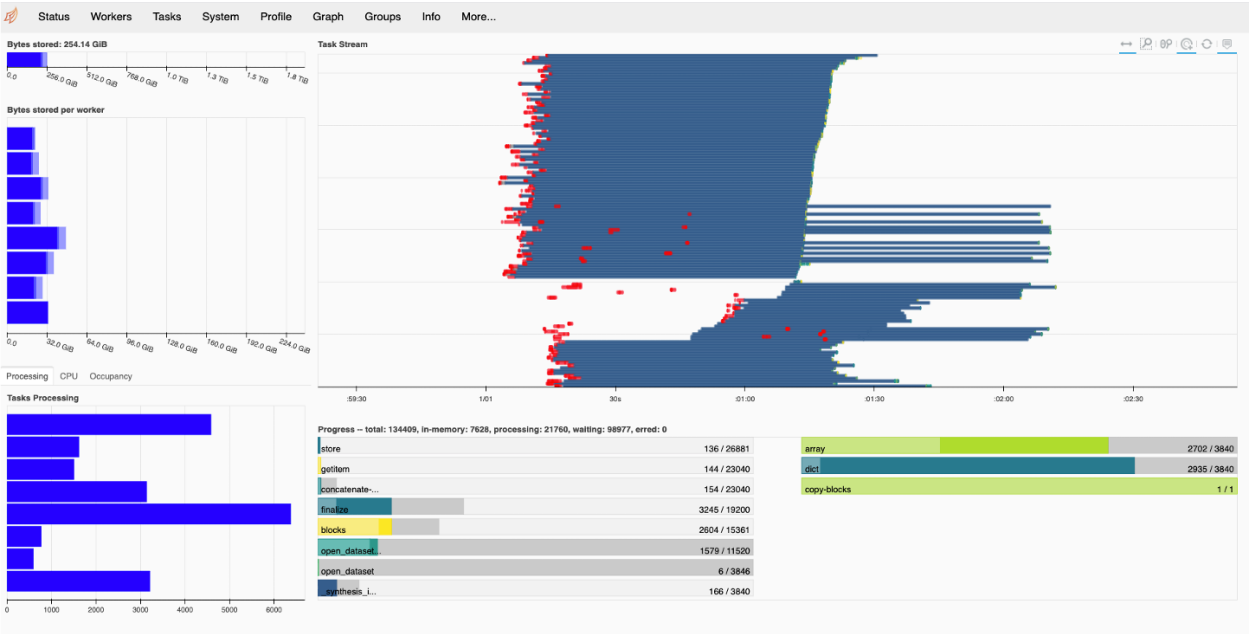
- Measurement Set Rows: 727272
- CNGI Shape (time, baseline, chan, pol): (2072, 351, 30720, 2)
- Image size (x,y,chan,pol): (1000, 1000, 30720, 2)
- Data Volume (vis.zarr and img.zarr): 2.5 TB

The CHILES dataset has a data volume over 8 times that of the largest ALMA dataset we benchmarked. For an 8 node (128 cores) run, the following results were achieved:

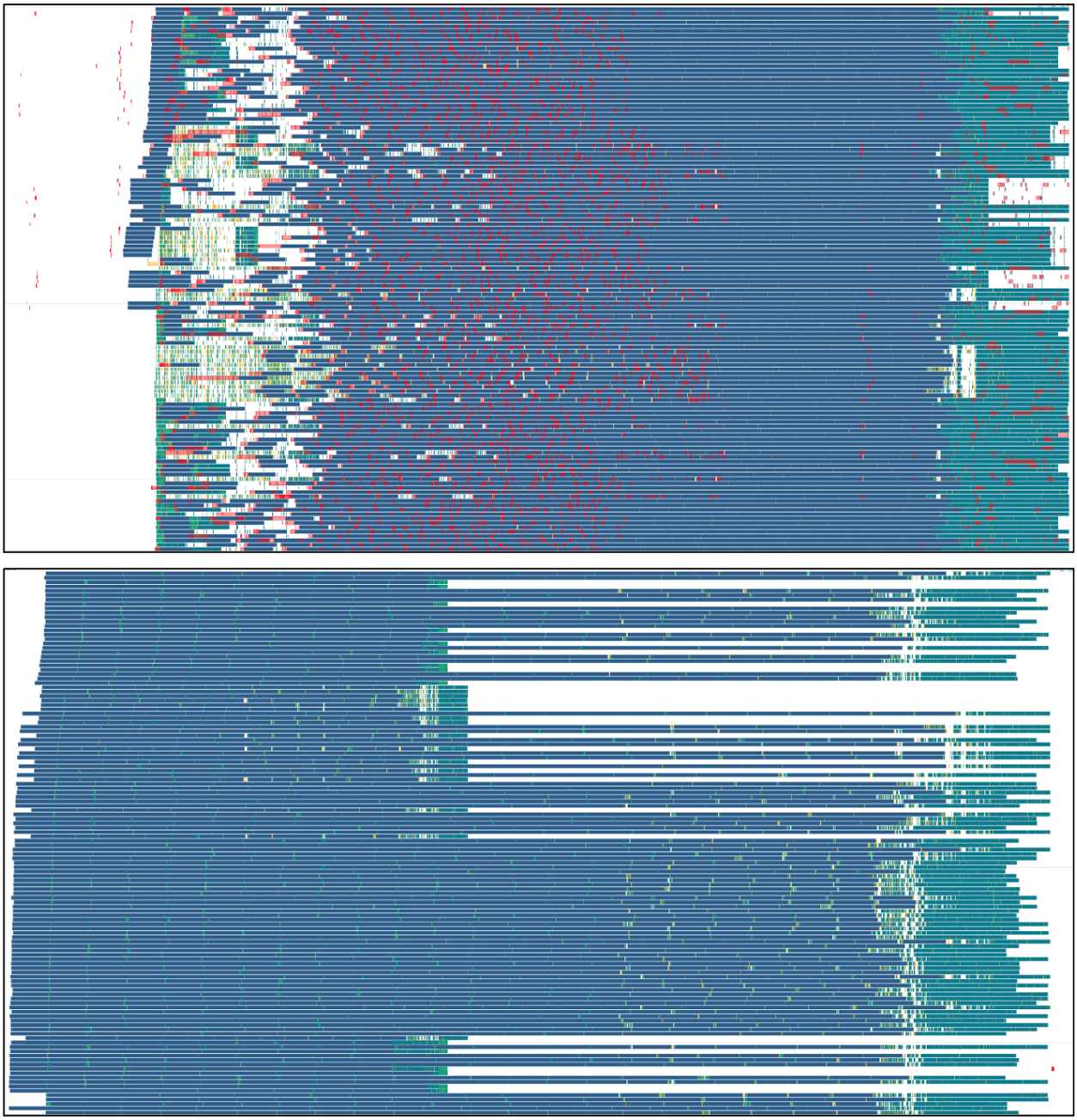
- CASA 6.2: 5 hours
- Prototype (3840 chunks): 45 minutes (6.7 x)
- Prototype (256 chunks): 14 hours (0.36 x)

The poor performance of the prototype for 256 chunks is due to the chunks being too large and intermittent data products spilling to disk. This can be seen in the Dask dashboard below (the gray bar shows the data spilled to disk):

When the number of chunks is increased to 3840 the Dask dashboard below shows that no spilling to disk occurs.

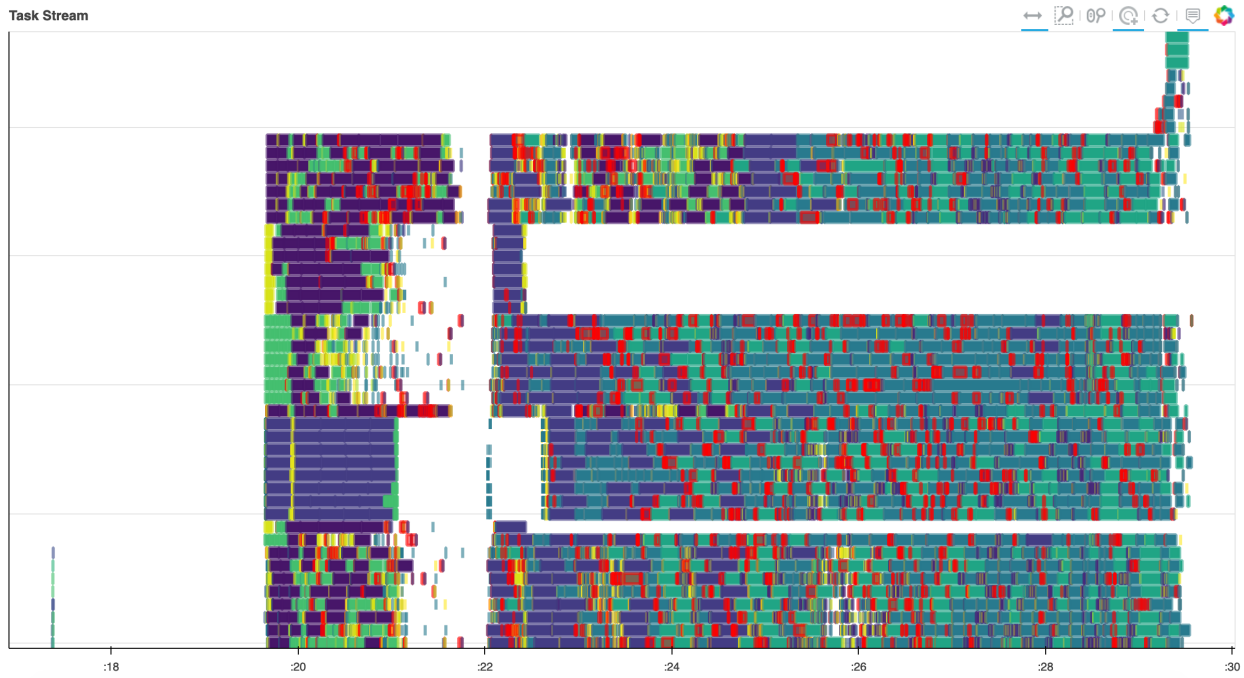


To reduce internode communication overhead, we modified the scheduler with the auto restrictor (AR) plugin developed by Jonathan Kenyon (https://github.com/dask/distributed/pull/4864/files). The first task stream is using the default Dask scheduler, and the communication overhead is shown in red. The second task stream is with the AR plugin enabled. While the communication overhead is eliminated the node work assignment is not balanced. Further modifications of the scheduler will be explored in the future.



9.5 Commercial Cloud

The total runtime curves for tests run on AWS show higher variance. One contributing factor that likely dominated this effect was the use of preemptible instances underlying the compute nodes running the worker configuration. For this same reason, some cloud-based test runs show decreased performance with increased scale. This is due to the preemption of nodes and associated redeployment by kubernetes, which sometimes constituted a large fraction of the total test runtime, as demonstrated by the task stream for the following test case. Note the horizontal white bar (signifying to tasks executed) shortly after graph execution begins, as well as some final tasks being assigned to a new node that came online after a few minutes (represented by the new "bar" of 8 rows at top right) in the following figure:

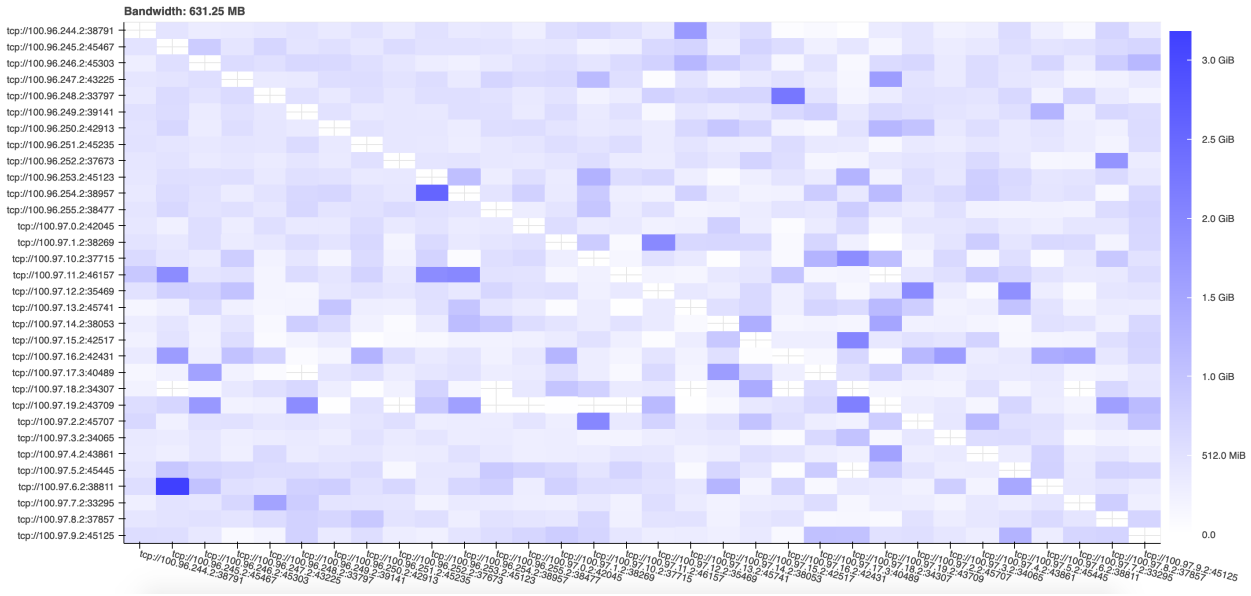


Qualitatively, failure rates were higher during tests of CASA on local HPC infrastructure than they were using dask on the cluster or cloud. The cube refactor shows a noticeable improvement in this area, but still worse than the prototype.

9.6 Profiling Results

Benchmarks performed using a single chunk size constitute a test of strong scaling (constant data volume, changing number of processors). Three of the projects used the mosaic gridder, the test of which consisted of multiple function calls composed into a rudimentary "pipeline". The other project used the standard gridder, with fewer separate function calls and thus relatively more time spent on compute.

The communication of data between workers constituted a relatively small proportion of the total runtime, and the distribution of data between workers was relatively uniform, at all horizontal scalings, with some hot spots beginning to present once tens of nodes were involved. This is demonstrated by the following figure, taken from the performance report of a representative test execution:



The time overhead associated with graph creation and task scheduling (approximately 100 ms per task for dask) grew as more nodes were introduced until eventually coming to represent a fraction of total execution time comparable to the computation itself, especially in the test cases with smaller data.

9.7 Reference Configurations

Dask profiling data were collected using the `performance_report <https://distributed.dask.org/en/latest/diagnosing-performance.html#performance-reports>`__ function in tests run both on-premises and in the commercial cloud.

Some values of the distributed configuration were modified from their defaults:

```
distributed:
  worker:
    # Fractions of worker memory at which we take action to avoid memory blowup
    # Set any of the lower three values to False to turn off the behavior entirely
    memory:
      target: 0.85  # fraction to stay below (default 0.60)
      spill: 0.92  # fraction at which we spill to disk (default 0.70)
      pause: 0.95  # fraction at which we pause worker threads (default 0.80)
```

Thread based parallelism in dependent libraries was disabled using environment variables `BLAS_NUM_THREADS`, `BLOSC_NOLOCK`, `MKL_NUM_THREADS`, and `OMP_NUM_THREADS`.

On-premises HPC cluster

- Test execution via Python scripts submitted to Moab scheduler and Torque resource manager with specifications documented internally
- Scheduling backend: `dask-jobqueue`
- I/O of visibility and image data via shared infiniband-interconnected lustre file system for access from on-premises high performance compute (HPC) nodes
- 16 threads per dask worker

- Compute via nodes from the cvpost batch queue with Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz with clock speed 1199.865 MHz and cache size 20480 KB.

Commercial cloud (AWS)

- Test execution via Jupyter notebooks running on a cloud deployment of the public dask-docker image (version 2021.3.0) backed by a Kubernetes cluster installed with `kops` (version 1.18.0), modified to include installation of version 0.0.83 of `cngi-prototype` and associated dependencies.
- Distributed scheduling backend: `dask.distributed`
- I/O of visibility and image data via Simple Storage Service (S3) object storage for access from commercial cloud Elastic Compute Cloud (EC2) nodes
- 8 threads per dask worker
- Compute via managed Kubernetes cluster backed by a variety of instance types all running on the current daily build of the Ubuntu 20.04 operating system. Cluster coordination service pods were run on a single dedicated `t3.small` instance. Jupyter notebook, dask scheduler, and etcd service pods were run on a single dedicated `m5dn.4xlarge` instance. Worker pods were run on a configured number of preemptible instances drawn from a pool composed of the following types: `m5.4xlarge`, `m5d.4xlarge`, `m5dn.4xlarge`, `r5.4xlarge`, `r4.4xlarge`,`m4.4xlarge`.

Hyperthreads exposed as vCPUs on the EC2 instances were disabled using the following shell script at instance launch:

```
spec:
  additionalUserData:
  - content: |
      #!/usr/bin/env bash
      for cpunum in $(cat /sys/devices/system/cpu/cpu*/topology/thread_siblings_list␣
↪| cut -s -d, -f2- | tr ',' '\n' | sort -un)
      do
        echo 0 > /sys/devices/system/cpu/cpu$cpunum/online
      done
    name: disable_hyperthreading.sh
    type: text/x-shellscript
  image:
```

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/verification.ipynb

CHAPTER

TEN

CASA MAPPING

CASA today has a complex interface that intertwines many use cases and target audiences, including interactive users and automated pipelines. The primary interface for most is the CASA task list. CASA tasks themselves range from simple functions to complex applications and are meant for a wide variety of purposes.

Here we map the current CASA task list to where the equivalent functionality will (eventually) live in the next generation software. In some cases this is not a one to one mapping, as tasks may be split in to multiple functions, or multiple tasks may be handled by a single function.

The future design thinking is to *separate functionality* into **CNGI**, **ngCASA**, and **Application** layers. The primary focus of this demonstration package is the CNGI layer, containing core mathematics and operations directly on datastructures themselves. Some preliminary ngCASA level design and prototyping work exists to build confidence that the CNGI level will suffice for next generation science performance. Very little application layer work has been done to date, but this will (eventually) be the layer that most users interact with, and tailored to specific use cases. Refer to the Introduction section for more information.

Tasks marked with N/A or "direct on xarray dataset" are no longer needed due to fundamental changes in data storage and access in CNGI.

1. accor - ngCASA TBD
2. apparentsens - ngCASA TBD
3. applycal - ngCASA TBD
4. asdmsummary - cngi.conversion TBD
5. bandpass - ngCASA TBD
6. blcal - ngCASA TBD
7. browsetable - ng Application TBD
8. calstat - ngCASA TBD
9. clearcal - ngCASA TBD
10. clearstat - **N/A**
11. concat - `cngi.vis.join_vis`
12. conjugatevis - **cngi direct on xarray dataset**
13. cvel - cngi.vis.regrid TBD
14. cvel2 - cngi.vis.regrid TBD
15. deconvolve - ngCASA TBD
16. delmod - **N/A**
17. exportasdm - cngi.conversion.save_asdm TBD

18. exportfits - cngi.conversion.save_image TBD
19. exportuvfits - cngi.conversion.save_ms TBD
20. feather - ngCASA TBD
21. fixplanets - ngCASA TBD
22. fixvis - cngi.vis.phase_shift TBD
23. flagcmd - ngCASA TBD
24. flagdata - ngCASA TBD
25. flagmanager - ngCASA TBD
26. fluxscale - ngCASA TBD
27. fringefit - ngCASA TBD
28. ft - ngCASA TBD
29. gaincal - ngCASA TBD
30. gencal - ngCASA TBD
31. hanningsmooth - `cngi.vis.chan_smooth`
32. imcollapse - **cngi direct on xarray dataset**
33. imcontsub - `cngi.image.cont_sub`
34. imdev - `cngi.image.statistics` + direct on xarray dataset
35. imfit - `cngi.image.fit_gaussian`
36. imhead - **cngi direct on xarray dataset**
37. imhistory - **cngi stored in xarray dataset attributes**
38. immath - **cngi direct on xarray dataset**
39. immoments - `cngi.image.moments`
40. impbcor - ngCASA TBD
41. importasdm - cngi.conversion.convert_asdm
42. importatca - cngi.conversion TBD
43. importfits - `cngi.conversion.convert_image`
44. importfitsidi - cngi.conversion TBD
45. importgmrt - cngi.conversion TBD
46. importmiriad - cngi.conversion TBD
47. importuvfits - cngi.conversion.convert_ms
48. importvla - cngi.conversion TBD
49. impv - cngi.image.posvel TBD
50. imrebin - `cngi.image.rebin`
51. imreframe - `cngi.image.reframe`
52. imregrid - cngi.image.regrid TBD
53. imsmooth - `cngi.image.smooth`

54. imstat - `cngi.image.statistics`
55. imsubimage - **cngi direct on xarray dataset**
56. imtrans - **cngi direct on xarray dataset**
57. imval - **cngi direct on xarray dataset**
58. imview - ng Application TBD
59. initweights - ngCASA TBD
60. listcal - **cngi direct on xarray dataset**
61. listfits - TBD
62. listhistory - **cngi stored in xarray dataset attributes**
63. listobs - **cngi direct on xarray dataset**
64. listpartition - **N/A**
65. listsdm - **cngi direct on xarray dataset**
66. listvis - **cngi direct on xarray dataset**
67. makemask - `cngi.image.mask`
68. mstransform - `cngi.vis.join_vis`, `cngi.vis.chan_average`, cngi.vis.regrid TBD, `cngi.vis.time_average`
69. msuvbin - ngCASA TBD
70. msview - ng Application TBD
71. nrobeamaverage - ngCASA TBD
72. partition - **N/A**
73. plotants - ng Application TBD
74. plotbandpass - ng Application TBD
75. plotcal - ng Application TBD
76. plotms - ng Application TBD
77. plotprofilemap - ng Application TBD
78. plotweather - ng Application TBD
79. polcal - ngCASA TBD
80. polfromgain - ngCASA TBD
81. predictcomp - ngCASA TBD
82. rerefant - ngCASA TBD
83. rmfit - cngi.image.rmfit TBD
84. rmtables - **N/A**
85. sdbaseline - cngi.vis.sd_fit TBD + direct on xarray dataset
86. sdcal - ngCASA TBD
87. sdfit - cngi.vis.sdfit TBD
88. sdfixscan - cngi.image.sd_fix_scan TBD

89. sdgaincal - ngCASA TBD
90. sdimaging - ngCASA TBD
91. sdintimaging - ng Application TBD
92. sdpolaverage - cngi.vis.sd_pol_average TBD
93. sdsidebandsplit - ngCASA TBD
94. sdsmooth - `cngi.vis.chan_smooth`
95. sdtimeaverage - `cngi.vis.time_average`
96. setjy - ngCASA TBD
97. simalma - ng Application TBD
98. simanalyze - ng Application TBD
99. simobserve - ng Application TBD
100. slsearch - cngi.external TBD
101. smoothcal - ngCASA TBD
102. specfit - `cngi.image.spec_fit`
103. specflux - cngi.image.spec_flux TBD
104. specsmooth - cngi.image.spec_smooth TBD
105. splattotable - cngi.external TBD
106. split - `cngi.vis.chan_average`, `cngi.vis.time_average` + `cngi.vis.split_dataset`
107. spxfit - cngi.image.spx_fit TBD
108. statwt - ngCASA TBD
109. tclean - ng Application TBD
110. tsdimaging - ng Application TBD
111. uvcontsub - `cngi.uv_cont_fit`
112. uvmodelfit - cngi.uv_model_fit TBD
113. uvsub - **cngi direct on xarray dataset**
114. viewer - CARTA
115. virtualconcat - **N/A**
116. vishead - **cngi direct on xarray dataset**
117. visstat - **cngi direct on xarray dataset**
118. widebandpbcor - ngCASA TBD
119. wvrgcal - ngCASA TBD

In the following sections we demonstrate how current CASA tasks may be satisfied in the new mapping to CNGI capabilities.

Note: This is a concept demonstration only and not meant to verify scientific accuracy

This section like most others may be run by readers in Google Colab using the provided link in the header. The first step in exdecution is to install necessary components and generate data to be used in the demonstration comparison to CASA6

[1]:
```
import os, warnings
warnings.simplefilter("ignore", category=RuntimeWarning)  # suppress warnings about␣
↪nan-slices
print("installing casa6 and cngi (takes a few minutes)...")
os.system("apt-get install libgfortran3")
os.system("pip install casatasks==6.3.0.48")
os.system("pip install casadata")
os.system("pip install cngi-prototype==0.0.92")

print("downloading MeasurementSet from CASAguide First Look at Imaging...")
!gdown -q --id 1N9QSs2Hbhi-BrEHx5PA54WigXt8GGgx1
!tar -xzf sis14_twhya_calibrated_flagged.ms.tar.gz
!mv sis14_twhya_calibrated_flagged.ms twhya.ms
print('converting ms...')
```

```
installing casa6 and cngi (takes a few minutes)...
downloading MeasurementSet from CASAguide First Look at Imaging...
converting ms...
```

[2]:
```
from casatasks import tclean
from cngi.conversion import convert_ms, convert_image
from cngi.image import implot
from cngi.vis import visplot
import numpy as np
import matplotlib.pyplot as plt

mxds = convert_ms('twhya.ms')

print("running tclean to generate an image")
tclean(vis='twhya.ms', imagename='twhya', field='5', spw='',
       specmode='cube', deconvolver='hogbom', nterms=1, imsize=[250,250], gridder=
↪'standard', cell=['0.1arcsec'],
       nchan=10, weighting='natural', threshold='0mJy', niter=100, interactive=False,␣
↪savemodel='modelcolumn',
       usemask='auto-multithresh')

image_xds = convert_image('twhya.image')
mxds = convert_ms('twhya.ms')  # reconvert to get MODEL_DATA col
```

```
Completed ddi 0  process time 22.91 s
Completed subtables  process time 1.06 s

running tclean to generate an image
converting Image...
processed image in 1.3743491 seconds
Completed ddi 0  process time 26.86 s
Completed subtables  process time 1.10 s
```

10.1 cngi.vis Module

Demonstration of CASA functions to be handled by the CNGI visibility module.

Note: this is a demonstration of mechanics only and is not intended for science

---

10.1.1 listobs

Note the mechanism for retrieving data from the MeasurementSet is fundamentally different, but the data itself is the same

[3]:
```
# CASA 6
from casatasks import listobs

listobs(vis='twhya.ms', listfile='obslist.txt', verbose=False, overwrite=True)
!cat obslist.txt
```

```
================================================================================
           MeasurementSet Name:  /content/twhya.ms      MS Version 2
================================================================================
   Observer: cqi     Project: uid://A002/X327408/X6f
Observation: ALMA(26 antennas)
Data records: 80563       Total elapsed time = 5647.68 seconds
   Observed from   19-Nov-2012/07:36:57.0   to   19-Nov-2012/09:11:04.7 (UTC)

Fields: 5
  ID   Code Name                RA               Decl           Epoch   SrcId      ␣
↪nRows
  0    none J0522-364           05:22:57.984648 -36.27.30.85128 J2000   0          ␣
↪4200
  2    none Ceres               06:10:15.950590 +23.22.06.90668 J2000   2          ␣
↪3800
  3    none J1037-295           10:37:16.079736 -29.34.02.81316 J2000   3          ␣
↪16000
  5    none TW Hya              11:01:51.796000 -34.42.17.36600 J2000   4          ␣
↪53161
  6    none 3c279               12:56:11.166576 -05.47.21.52464 J2000   5          ␣
↪3402
Spectral Windows:  (1 unique spectral windows and 1 unique polarization setups)
  SpwID  Name                           #Chans   Frame   Ch0(MHz)  ChanWid(kHz) ␣
↪TotBW(kHz) CtrFreq(MHz) BBC Num  Corrs
  0      ALMA_RB_07#BB_2#SW-01#FULL_RES    384   TOPO  372533.086       610.352   ␣
↪234375.0 372649.9688        2  XX  YY
Antennas: 21 'name'='station'
   ID=   1-4: 'DA42'='A050', 'DA44'='A068', 'DA45'='A070', 'DA46'='A067',
   ID=   5-9: 'DA48'='A046', 'DA49'='A029', 'DA50'='A045', 'DV02'='A077',
   ID= 10-15: 'DV05'='A082', 'DV06'='A037', 'DV08'='A021', 'DV10'='A071',
   ID= 16-19: 'DV13'='A072', 'DV15'='A074', 'DV16'='A069', 'DV17'='A138',
   ID= 20-24: 'DV18'='A053', 'DV19'='A008', 'DV20'='A020', 'DV22'='A011',
   ID= 25-25: 'DV23'='A007'
```

[4]:
```
# CNGI
#print('Observation ', mxds.OBSERVATION.compute().data_vars)
#print('Field ', mxds.FIELD.compute().data_vars)
```

(continues on next page)

(continued from previous page)

```
#print('Spectral Window ', mxds.SPECTRAL_WINDOW.compute().data_vars)
#print('Source ', mxds.SOURCE.compute().data_vars)
#print('Antenna ', mxds.ANTENNA.compute().data_vars)
print(mxds.dims)
print(mxds.coords)
```

```
Frozen(SortedKeysDict({'antenna_ids': 26, 'field_ids': 7, 'feed_ids': 26,
↪'observation_ids': 1, 'polarization_ids': 1, 'source_ids': 5, 'spw_ids': 1, 'state_
↪ids': 20}))
Coordinates:
  * antenna_ids       (antenna_ids) int64 0 1 2 3 4 5 6 ... 19 20 21 22 23 24 25
    antennas          (antenna_ids) <U16 'DA41' 'DA42' 'DA44' ... 'DV22' 'DV23'
  * field_ids         (field_ids) int64 0 1 2 3 4 5 6
    fields            (field_ids) <U9 'J0522-364' 'J0539+145' ... '3c279'
  * feed_ids          (feed_ids) int64 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  * observation_ids   (observation_ids) int64 0
    observations      (observation_ids) <U23 'uid://A002/X327408/X6f'
  * polarization_ids  (polarization_ids) int64 0
  * source_ids        (source_ids) int64 0 1 2 3 4
    sources           (source_ids) <U9 'J0522-364' 'Ceres' ... 'TW Hya' '3c279'
  * spw_ids           (spw_ids) int64 0
  * state_ids         (state_ids) int64 0 1 2 3 4 5 6 7 ... 13 14 15 16 17 18 19
```

10.1.2 listvis

Note the mechanism for retrieving data from the MeasurementSet is fundamentally different, but the data itself is the same

[5]:
```
# CASA 6
from casatasks import listvis

os.system("rm -fr vislist.txt")
listvis(vis='twhya.ms', field="J1037-295", timerange='<07:53:00', antenna='DA46',
        spw='*:30~40', datacolumn='data', listfile='vislist.txt')
!tail -n 18 vislist.txt
```

```
Units of columns are: Date/Time(YYMMDD/HH:MM:SS UT), UVDist(wavelength), Phase(deg),␣
↪UVW(m)
FIELD: 3
SPW: 0
Date/Time:                              XX:                   YY:
2012/11/19/      Intrf UVDist  Chn     Amp     Phs  Wt F    Amp     Phs  Wt F        ␣
↪U         V         W
------------|---------|------|----|--------------------|--------------------|--------
↪-|---------|---------|
  07:52:57.2 DA46-DV23  65292   30: 14.690   -18.7   9    6.279    40.7  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   31:  1.951    62.0   9    1.091   132.5  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   32: 12.303    -5.4   9    1.837    83.7  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   33: 32.372   -63.7   9    9.889   -18.0  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   34: 13.741    61.8   9   19.342   -45.3  17       44.
↪17     28.46     43.89
```

(continues on next page)

(continued from previous page)

```
  07:52:57.2 DA46-DV23  65292   35: 14.456   109.7   9   10.931   -75.3  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   36: 10.487     4.9   9   11.387   -79.9  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   37:  9.204  -136.4   9    5.963  -166.3  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   38: 33.276     1.9   9    4.406    67.3  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   39: 16.079   108.8   9    8.067   172.0  17       44.
↪17     28.46     43.89
  07:52:57.2 DA46-DV23  65292   40: 16.795   -20.8   9    9.997    96.8  17       44.
↪17     28.46     43.89
------------|---------|------|----|--------------------|--------------------|--------
↪-|---------|---------|
```

[6]:
```
# CNGI
field_id = mxds.field_ids[np.where(mxds.fields == "J1037-295")][0]
selection = mxds.xds0.isel(chan=range(30,40)).sel(time='2012-11-19T07:52').where(mxds.
↪xds0.FIELD_ID == field_id, drop=True).compute()
print(selection)
```

```
<xarray.Dataset>
Dimensions:          (baseline: 190, chan: 10, pol: 2, pol_id: 1, spw_id: 1, time: 3,␣
↪uvw_index: 3)
Coordinates:
  * time             (time) datetime64[ns] 2012-11-19T07:52:45.072000504 ... 2...
  * baseline         (baseline) int64 0 1 2 3 4 5 6 ... 201 202 203 207 208 209
  * chan             (chan) float64 3.726e+11 3.726e+11 ... 3.726e+11 3.726e+11
    chan_width       (chan) float64 6.104e+05 6.104e+05 ... 6.104e+05 6.104e+05
    effective_bw     (chan) float64 6.104e+05 6.104e+05 ... 6.104e+05 6.104e+05
  * pol              (pol) int64 9 12
  * pol_id           (pol_id) int64 0
    resolution       (chan) float64 6.104e+05 6.104e+05 ... 6.104e+05 6.104e+05
  * spw_id           (spw_id) int64 0
    field_ids        int64 3
    fields           <U9 'J1037-295'
Dimensions without coordinates: uvw_index
Data variables: (12/18)
    ANTENNA1         (baseline, time) float64 1.0 1.0 1.0 1.0 ... 24.0 24.0 24.0
    ANTENNA2         (baseline, time) float64 2.0 2.0 2.0 3.0 ... 25.0 25.0 25.0
    ARRAY_ID         (time, baseline) float64 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    DATA             (time, baseline, chan, pol) complex128 (-2.60023474693298...
    DATA_WEIGHT      (time, baseline, chan, pol) float64 12.39 21.67 ... 14.35
    EXPOSURE         (time, baseline) float64 6.048 6.048 6.048 ... 6.048 6.048
    ...               ...
    OBSERVATION_ID   (time, baseline) float64 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    PROCESSOR_ID     (time, baseline) float64 2.0 2.0 2.0 2.0 ... 2.0 2.0 2.0 2.0
    SCAN_NUMBER      (time, baseline) float64 10.0 10.0 10.0 ... 10.0 10.0 10.0
    STATE_ID         (time, baseline) float64 8.0 8.0 8.0 8.0 ... 8.0 8.0 8.0 8.0
    TIME_CENTROID    (time, baseline) float64 4.86e+09 4.86e+09 ... 4.86e+09
    UVW              (time, baseline, uvw_index) float64 115.4 -56.68 ... 22.89
Attributes: (12/14)
    assoc_nature:      ['', '', '', '', '', '', '', '', '', '', '', '', '', ''...
    bbc_no:            2
    corr_product:      [[0, 0], [1, 1]]
    data_groups:       [{'0': {'data': 'DATA', 'flag': 'FLAG', 'id': '0', 'uvw...
    freq_group:        0
```

(continues on next page)

(continued from previous page)

```
    freq_group_name:
    ...                 ...
    name:               ALMA_RB_07#BB_2#SW-01#FULL_RES
    net_sideband:       2
    num_chan:           384
    num_corr:           2
    ref_frequency:      372533086425.9812
    total_bandwidth:    234375000.0
```

10.1.3 concat

[7]:
```
# CASA 6
from casatasks import concat, split

# first we need to create two ms's, then concat them
split('twhya.ms', 'ms1.ms', field='J0522-364', datacolumn='DATA')
split('twhya.ms', 'ms2.ms', field='Ceres', datacolumn='DATA')
concat(['ms1.ms', 'ms2.ms'], 'concatms.ms')
listobs(vis='concatms.ms', listfile='obslist.txt', verbose=False, overwrite=True)
!cat obslist.txt
```

```
================================================================================
           MeasurementSet Name:  /content/concatms.ms      MS Version 2
================================================================================
   Observer: cqi     Project: uid://A002/X327408/X6f
Observation: ALMA(26 antennas)
Data records: 8000       Total elapsed time = 604.272 seconds
   Observed from   19-Nov-2012/07:36:57.0   to   19-Nov-2012/07:47:01.2 (UTC)

Fields: 2
  ID   Code Name                RA               Decl           Epoch   SrcId      ␣
↪nRows
  0    none J0522-364           05:22:57.984648 -36.27.30.85128 J2000   0          ␣
↪4200
  1    none Ceres               06:10:15.950590 +23.22.06.90668 J2000   1          ␣
↪3800
Spectral Windows:  (1 unique spectral windows and 1 unique polarization setups)
  SpwID  Name                           #Chans   Frame   Ch0(MHz)  ChanWid(kHz) ␣
↪TotBW(kHz) CtrFreq(MHz) BBC Num  Corrs
  0      ALMA_RB_07#BB_2#SW-01#FULL_RES    384   TOPO  372533.086       610.352   ␣
↪234375.0 372649.9688        2  XX  YY
Antennas: 21 'name'='station'
   ID=   1-4: 'DA42'='A050', 'DA44'='A068', 'DA45'='A070', 'DA46'='A067',
   ID=   5-9: 'DA48'='A046', 'DA49'='A029', 'DA50'='A045', 'DV02'='A077',
   ID= 10-15: 'DV05'='A082', 'DV06'='A037', 'DV08'='A021', 'DV10'='A071',
   ID= 16-19: 'DV13'='A072', 'DV15'='A074', 'DV16'='A069', 'DV17'='A138',
   ID= 20-24: 'DV18'='A053', 'DV19'='A008', 'DV20'='A020', 'DV22'='A011',
   ID= 25-25: 'DV23'='A007'
```

[8]:
```
# CNGI
from cngi.vis import join_dataset
```

(continues on next page)

(continued from previous page)

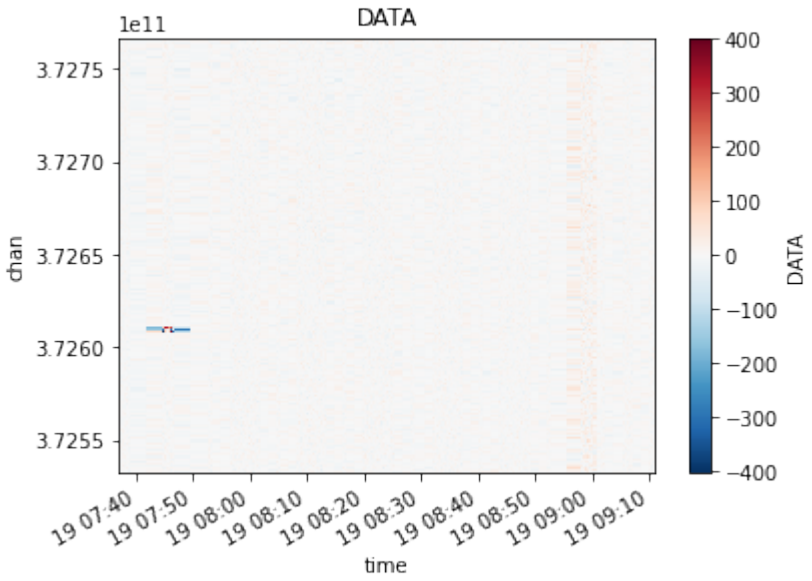
```
mxds1 = convert_ms('ms1.ms')
mxds2 = convert_ms('ms2.ms')
jmxds = join_dataset(mxds1, mxds2)

print(jmxds.dims)
print(jmxds.coords)
```

```
Completed ddi 0  process time 1.64 s
Completed subtables  process time 1.06 s

Completed ddi 0  process time 1.61 s
Completed subtables  process time 1.07 s

Warning: reference value -1 in subtable WEATHER does not exist in ANTENNA.antenna_id!
Frozen(SortedKeysDict({'antenna_ids': 26, 'field_ids': 2, 'feed_ids': 26,
↪'observation_ids': 1, 'polarization_ids': 1, 'source_ids': 2, 'spw_ids': 1, 'state_
↪ids': 20}))
Coordinates:
  * antenna_ids       (antenna_ids) int64 0 1 2 3 4 5 6 ... 19 20 21 22 23 24 25
    antennas          (antenna_ids) <U16 'DA41' 'DA42' 'DA44' ... 'DV22' 'DV23'
  * field_ids         (field_ids) int64 0 1
    fields            (field_ids) object 'J0522-364' 'Ceres'
  * feed_ids          (feed_ids) int64 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  * observation_ids   (observation_ids) int64 0
    observations      (observation_ids) <U23 'uid://A002/X327408/X6f'
  * polarization_ids  (polarization_ids) int64 0
  * source_ids        (source_ids) int64 0 1
    sources           (source_ids) object 'J0522-364' 'J1037-295'
  * spw_ids           (spw_ids) int64 0
  * state_ids         (state_ids) int64 0 1 2 3 4 5 6 7 ... 13 14 15 16 17 18 19
```

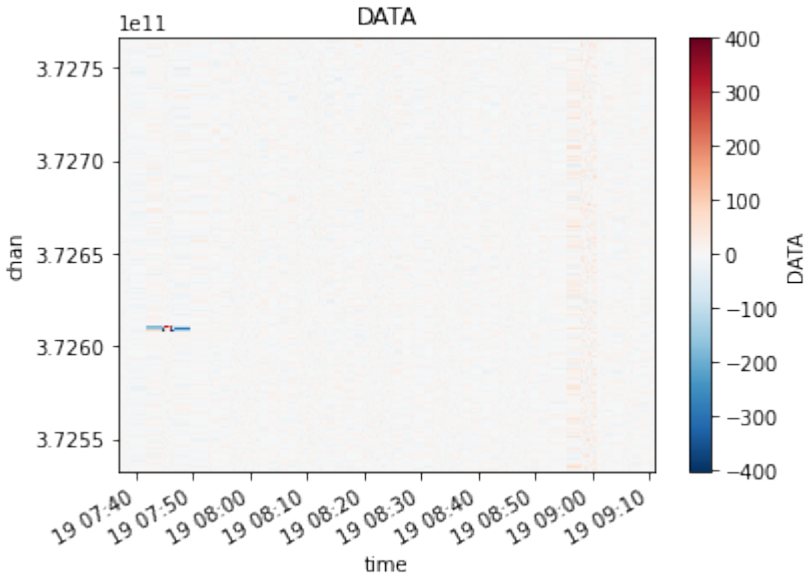
10.1.4 conjugatevis

[9]:
```
# CASA 6
from casatasks import conjugatevis

conjugatevis(vis='twhya.ms', outputvis='casa6.conj.ms', overwrite=True)

casa_xds = convert_ms('casa6.conj.ms').xds0
visplot(casa_xds.DATA.imag[:,100,:,0], axis=['time','chan'])
```

```
Completed ddi 0  process time 27.06 s
Completed subtables  process time 1.10 s
```



[10]:
```
# CNGI
cngi_xda = mxds.xds0.DATA.conj()

visplot(cngi_xda.imag[:,100,:,0], axis=['time','chan'])
```



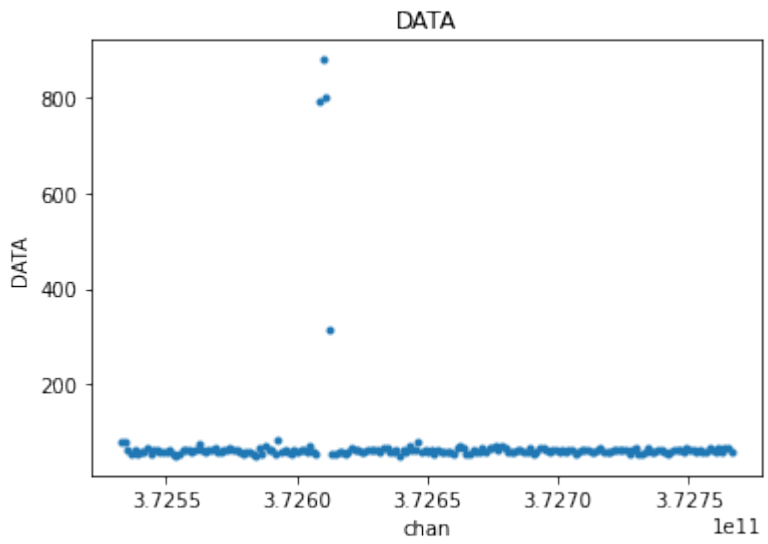
[11]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_xds.DATA - cngi_xda).values)))
```

```
max delta :  0.0
```

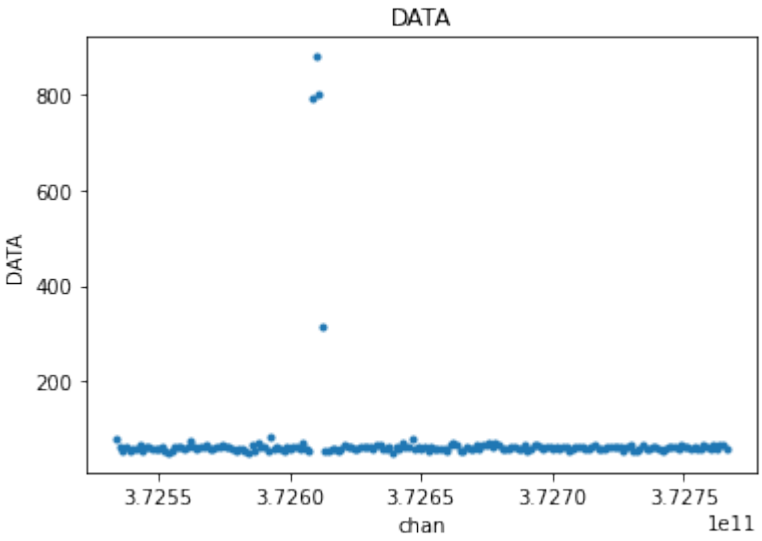
10.1.5 hanningsmooth

[12]:
```
# CASA6
from casatasks import hanningsmooth

hanningsmooth(vis='twhya.ms', outputvis='casa6.smooth.ms', datacolumn='data')

casa_xds = convert_ms('casa6.smooth.ms').xds0
visplot(casa_xds.DATA, axis='chan')
```

```
Completed ddi 0  process time 22.05 s
Completed subtables  process time 1.06 s
```



[13]:
```
# CNGI
from cngi.vis import apply_flags, chan_smooth

cngi_xds = chan_smooth(mxds, 'xds0').xds0

visplot(cngi_xds.DATA, axis='chan')
```



[14]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_xds.DATA - cngi_xds.DATA).values)))
```

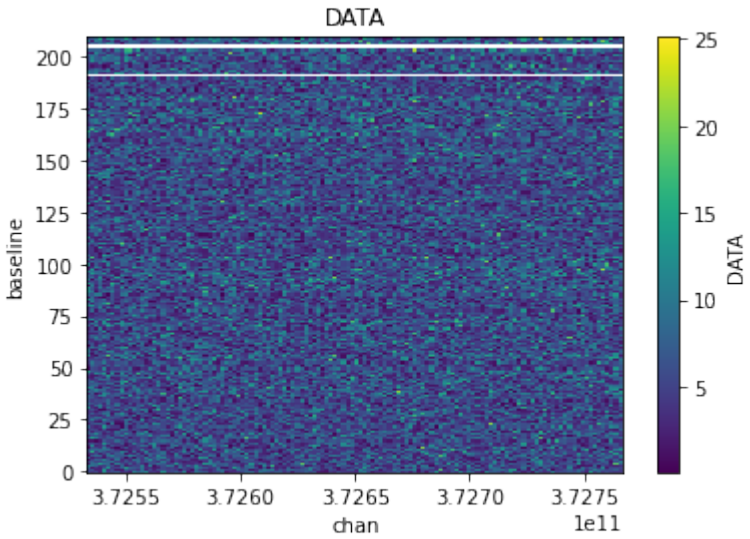
```
max delta :  6.104260637590507e-05
```

10.1.6 mstransform

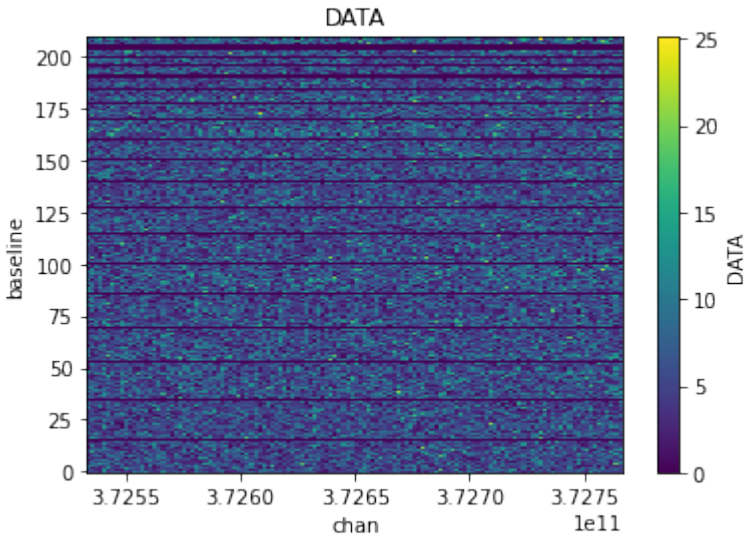
chanaverage

[15]:
```
# CASA 6
from casatasks import mstransform

os.system("rm -fr casa6.avg.ms")
mstransform(vis='twhya.ms', outputvis='casa6.avg.ms', datacolumn='DATA',␣
↪chanaverage=True, chanbin=3)

casa_xds = convert_ms('casa6.avg.ms').xds0
visplot(casa_xds.DATA[100,:,:,0], axis=['chan', 'baseline'])
```

```
Completed ddi 0  process time 8.50 s
Completed subtables  process time 1.04 s
```



[16]:
```
# CNGI
from cngi.vis import chan_average

cngi_xds = chan_average(mxds, 'xds0', width=3).xds0

visplot(cngi_xds.DATA[100,:,:,0], axis=['chan', 'baseline'])
```



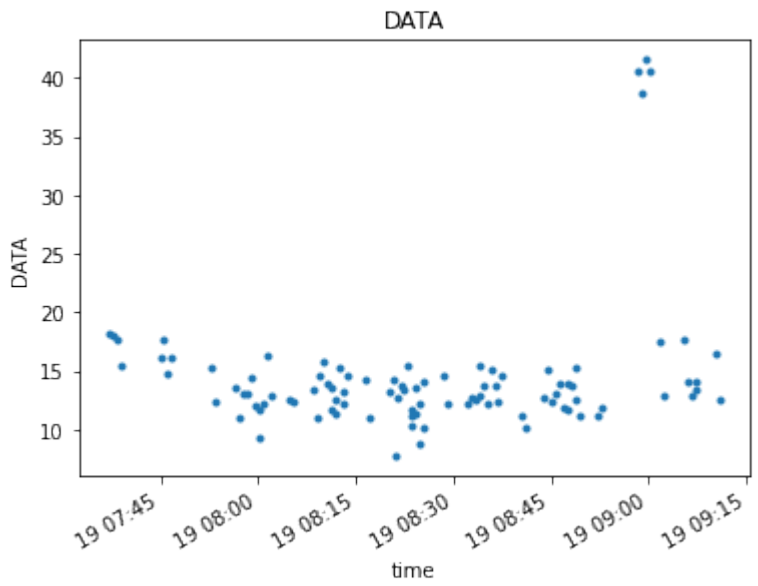
[17]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_xds.DATA[100,:,:,0] - cngi_xds.DATA[100,:
↪,:,0]).values)))
```

```
max delta :  1.5973847098419519e-06
```

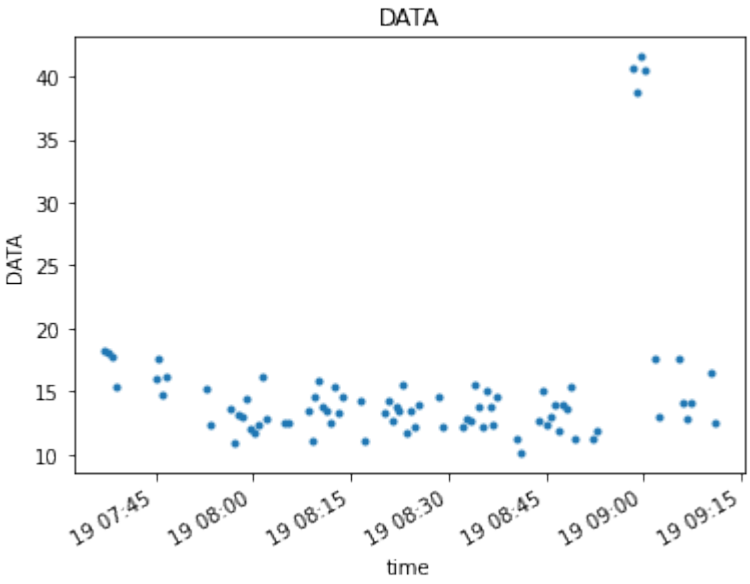
timeaverage

[18]:
```
# CASA 6
from casatasks import mstransform

os.system("rm -fr casa6.tavg.ms")
mstransform(vis='twhya.ms', outputvis='casa6.tavg.ms', datacolumn='DATA',␣
↪timeaverage=True, timebin='25s', timespan='state')

casa_xds = convert_ms('casa6.tavg.ms').xds0
visplot(casa_xds.DATA[:,:,100,0], axis='time')
```

```
Completed ddi 0  process time 4.47 s
Completed subtables  process time 1.04 s
```



[19]:
```
# CNGI
from cngi.vis import time_average

mxds2 = mxds.assign_attrs({'xds1':mxds.xds0.chunk(400, 210, 64, 1)}) # increase chunk␣
↪size to avoid warnings
cngi_xds = time_average(mxds2, 'xds1', bin=5, span='state').xds1

visplot(cngi_xds.DATA[:,:,100,0], axis='time')
```



[20]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_xds.DATA[:,:,100,0] - cngi_xds.DATA[:,:,
↪100,0]).values)))
```

```
max delta :  5.173171442074411e-06
```

10.1.7 sdsmooth - TBD

likely to be demonstrated as satisfied by cngi.vis.chan_smooth

---

10.1.8 sdtimeaverage - TBD

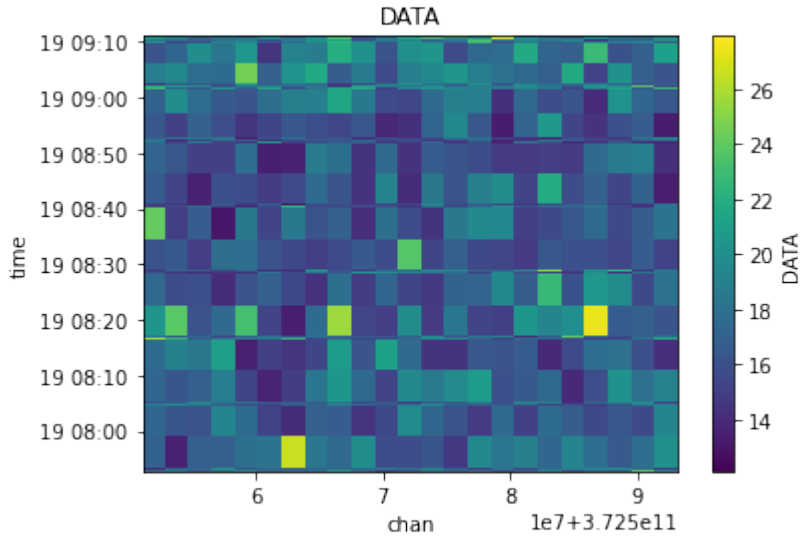
likely to be demonstrated as satisfied by cngi.vis.time_average

---

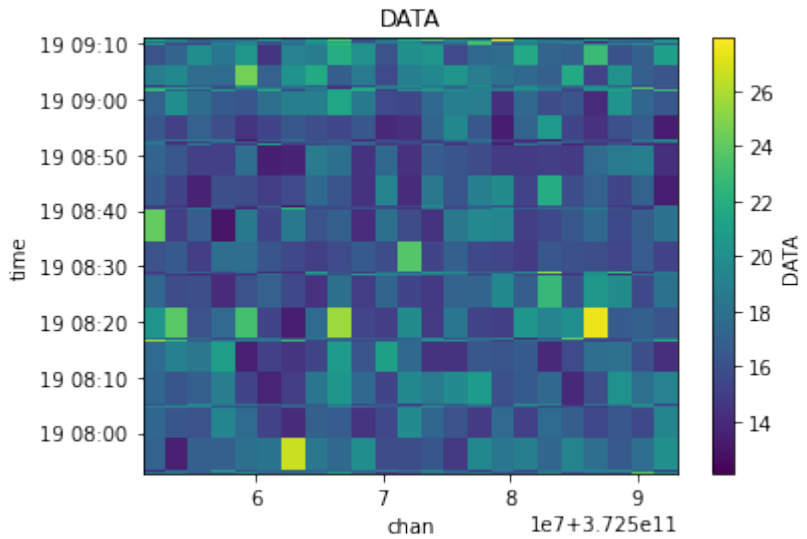
10.1.9 split

[21]:
```
# CASA 6
from casatasks import split

os.system("rm -fr casa6.split.ms")
split(vis='twhya.ms', outputvis='casa6.split.ms', datacolumn='DATA', spw='*:30~100',␣
↪field='3', width=3)

casa_xds = convert_ms('casa6.split.ms').xds0
visplot(casa_xds.DATA, axis=['chan', 'time'])
```

```
Completed ddi 0  process time 0.81 s
Completed subtables  process time 1.06 s
```



[22]:
```
# CNGI
from cngi.vis import chan_average

selection = mxds.xds0.isel(chan=range(30,101)).where(mxds.xds0.FIELD_ID == 3,␣
↪drop=True)
cngi_xds = chan_average(mxds.assign_attrs({'xds1':selection}), 'xds1', width=3).xds1

visplot(cngi_xds.DATA, axis=['chan', 'time'])
```



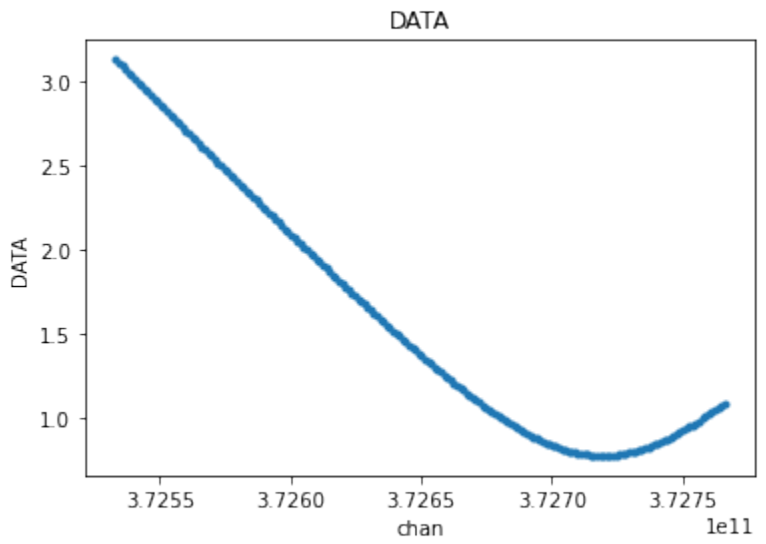
[23]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_xds.DATA - cngi_xds.DATA).values)))
```

```
max delta :  2.132480599880018e-06
```

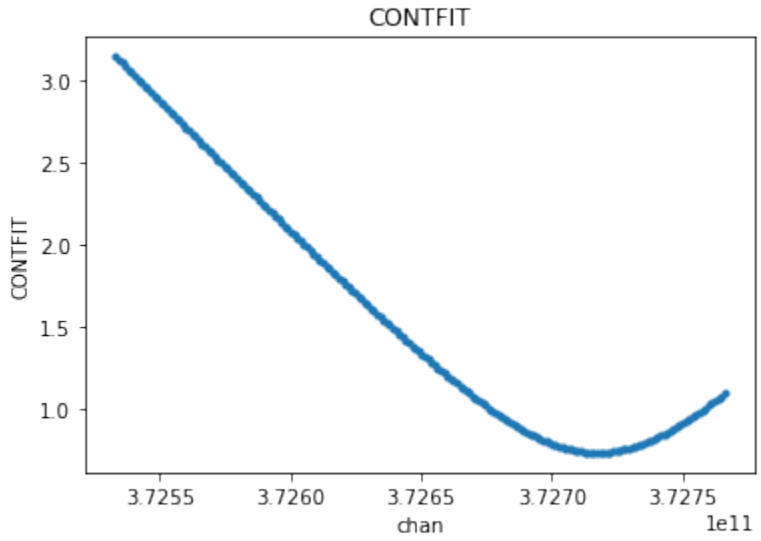
10.1.10 uvcontsub

[24]:
```
# CASA 6
from casatasks import uvcontsub

uvcontsub(vis='twhya.ms', field="5", combine='scan', fitspw='0:200~300',␣
↪excludechans=True, fitorder=1, want_cont=True)

casa_cont_xds = convert_ms('twhya.ms.cont').xds0

visplot(casa_cont_xds.DATA[10,10,:,0], axis=['chan'])
```

```
Completed ddi 0  process time 12.90 s
Completed subtables  process time 1.31 s
```



[25]:
```
# CNGI
from cngi.vis import uv_cont_fit

selection = mxds.xds0.where(mxds.xds0.FIELD_ID == 5, drop=True)
cngi_xds = uv_cont_fit(mxds.assign_attrs({'xds1':selection}), 'xds1', source='DATA',␣
↪target='CONTFIT', fitorder=1, excludechans=list(range(200,300))).xds1

visplot(cngi_xds.CONTFIT[10,10,:,0], axis=['chan'])
```



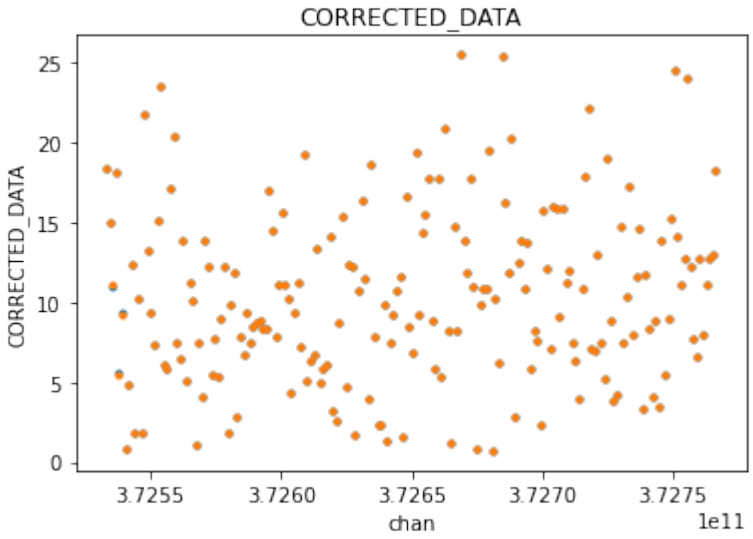
[26]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_cont_xds.DATA - cngi_xds.CONTFIT).
↪values)))
```

```
max delta :  0.44718851662412784
```

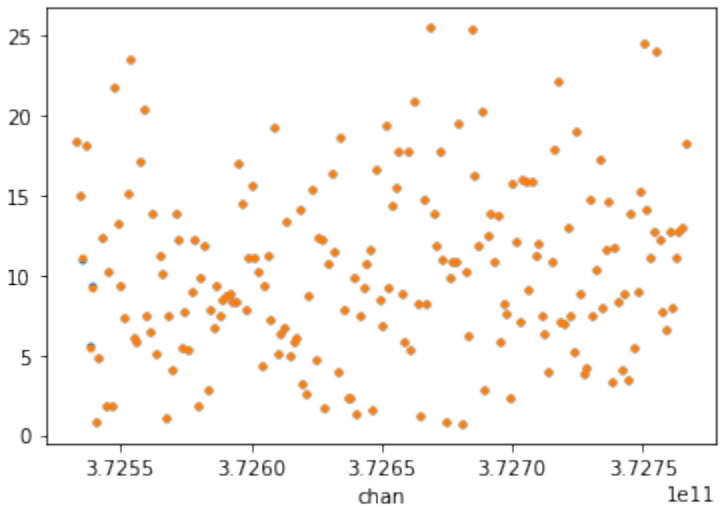
10.1.11 uvsub

[27]:
```
# CASA 6
from casatasks import uvsub

os.system('cp -r twhya.ms uvsub.ms')
uvsub('uvsub.ms')

casa_xds = convert_ms('uvsub.ms').xds0

visplot(casa_xds.DATA[310,10,:,0], axis=['chan'], drawplot=False)
visplot(casa_xds.CORRECTED_DATA[310,10,:,0], axis=['chan'], overplot=True)
```

```
Completed ddi 0  process time 48.04 s
Completed subtables  process time 1.20 s
```



[28]:
```
# CNGI
from cngi.vis import apply_flags

cngi_xda = mxds.xds0.DATA - mxds.xds0.MODEL_DATA

visplot(mxds.xds0.DATA[310,10,:,0], axis=['chan'], drawplot=False)
visplot(cngi_xda[310,10,:,0], axis=['chan'], overplot=True)
```



[29]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_xds.CORRECTED_DATA - cngi_xda).values)))
```

```
max delta :  3.0517578125e-05
```

10.1.12 vishead

Note the mechanism for retrieving data from the MeasurementSet is fundamentally different, but the data itself is the same

[30]:
```
# CASA 6
from casatasks import vishead

vishead(vis='twhya.ms', mode='list')
```

[30]:
```
{'field': (array(['J0522-364', 'J0539+145', 'Ceres', 'J1037-295', 'TW Hya', 'TW Hya',
         '3c279'], dtype='<U16'), {}),
 'freq_group_name': (array([''], dtype='<U16'), {}),
 'observer': (array(['cqi'], dtype='<U16'), {}),
 'project': (array(['uid://A002/X327408/X6f'], dtype='<U23'), {}),
 'release_date': (array([0.]),
  {'MEASINFO': {'Ref': 'UTC', 'type': 'epoch'},
   'QuantumUnits': array(['s'], dtype='<U16')}),
 'schedule': ({'r1': array([['SchedulingBlock uid://A002/X327408/X73'],
          ['ExecBlock uid://A002/X554543/X207']], dtype='<U39')}, {}),
 'schedule_type': (array(['ALMA'], dtype='<U16'), {}),
 'spw_name': (array(['ALMA_RB_07#BB_2#SW-01#FULL_RES'], dtype='<U31'), {}),
 'telescope': (array(['ALMA'], dtype='<U16'), {})}
```

[31]:
```
# CNGI
print(mxds)
```

```
<xarray.Dataset>
Dimensions:           (antenna_ids: 26, feed_ids: 26, field_ids: 7, observation_ids:␣
↪1, polarization_ids: 1, source_ids: 5, spw_ids: 1, state_ids: 20)
Coordinates:
  * antenna_ids       (antenna_ids) int64 0 1 2 3 4 5 6 ... 19 20 21 22 23 24 25
    antennas          (antenna_ids) <U16 'DA41' 'DA42' 'DA44' ... 'DV22' 'DV23'
  * field_ids         (field_ids) int64 0 1 2 3 4 5 6
    fields            (field_ids) <U9 'J0522-364' 'J0539+145' ... '3c279'
  * feed_ids          (feed_ids) int64 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0
  * observation_ids   (observation_ids) int64 0
    observations      (observation_ids) <U23 'uid://A002/X327408/X6f'
  * polarization_ids  (polarization_ids) int64 0
  * source_ids        (source_ids) int64 0 1 2 3 4
    sources           (source_ids) <U9 'J0522-364' 'Ceres' ... 'TW Hya' '3c279'
  * spw_ids           (spw_ids) int64 0
  * state_ids         (state_ids) int64 0 1 2 3 4 5 6 7 ... 13 14 15 16 17 18 19
Data variables:
    *empty*
Attributes: (12/17)
    xds0:             <xarray.Dataset>\nDimensions:         (baseline: 210, c...
    ANTENNA:          <xarray.Dataset>\nDimensions:        (antenna_id: 26, d...
    ASDM_ANTENNA:     <xarray.Dataset>\nDimensions:         (d0: 26, d1: 3)\n...
    ASDM_CALWVR:      <xarray.Dataset>\nDimensions:            (d0: 702, d1: ...
    ASDM_RECEIVER:    <xarray.Dataset>\nDimensions:           (d0: 26, d1: 2,...
    ASDM_STATION:     <xarray.Dataset>\nDimensions:    (d0: 28, d1: 3)\nDimen...
    ...               ...
    POLARIZATION:     <xarray.Dataset>\nDimensions:       (d0: 1, d1: 2, d2: ...
    PROCESSOR:        <xarray.Dataset>\nDimensions:   (d0: 3)\nCoordinates:\n...
    SOURCE:           <xarray.Dataset>\nDimensions:             (d0: 5, d1: 2...
    SPECTRAL_WINDOW:  <xarray.Dataset>\nDimensions:             (d1: 384, d2:...
    STATE:            <xarray.Dataset>\nDimensions:   (state_id: 20)\nCoordin...
    WEATHER:          <xarray.Dataset>\nDimensions:                 (d0: 385,...
```

10.1.13 visstat

[32]:
```
# CASA 6
from casatasks import visstat

casa_dict = visstat(vis='twhya.ms', axis='real', datacolumn='data')['DATA_DESC_ID=0']
print(casa_dict['max'], casa_dict['stddev'])
```

```
106.0803451538086 8.556056451642778
```

[33]:
```
# CNGI
from cngi.vis import apply_flags

cngi_max = apply_flags(mxds, 'xds0').xds0.DATA.real.max().values
cngi_std = apply_flags(mxds, 'xds0').xds0.DATA.real.std().values

print(cngi_max, cngi_std)
```

```
106.0803451538086 8.556056382448858
```

[34]:
```
# Delta
print('max delta : ', np.max(np.abs([casa_dict['max'] - cngi_max, casa_dict['stddev']␣
↪- cngi_std])))
```

```
max delta :  6.919391992710189e-08
```

10.2 cngi.image Module

Demonstration of CASA functions to be handled by the CNGI image module.

Note: this is a demonstration of mechanics only and is not intended for science

10.2.1 imhead

Note the mechanism for retrieving data from the image is fundamentally different, but the data itself is the same

[35]:
```
# CASA6
from casatasks import imhead

casa_dict = imhead(imagename='twhya.image', mode='summary')
casa_dict
```

[35]:
```
{'axisnames': array(['Right Ascension', 'Declination', 'Stokes', 'Frequency'],
       dtype='<U16'),
 'axisunits': array(['rad', 'rad', '', 'Hz'], dtype='<U16'),
 'defaultmask': 'mask0',
 'hasmask': True,
 'imagetype': 'Intensity',
 'incr': array([-4.84813681e-07,  4.84813681e-07,  1.00000000e+00,  6.10330159e+05]),
 'masks': array(['mask0'], dtype='<U16'),
 'messages': array([], dtype='<U16'),
```

(continues on next page)

(continued from previous page)

```
 'ndim': 4,
 'perplanebeams': {'beams': {'*0': {'*0': {'major': {'unit': 'arcsec',
      'value': 0.6526992917060852},
     'minor': {'unit': 'arcsec', 'value': 0.5043594837188721},
     'positionangle': {'unit': 'deg', 'value': -65.8895263671875}}},
   '*1': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526992917060852},
     'minor': {'unit': 'arcsec', 'value': 0.5043594837188721},
     'positionangle': {'unit': 'deg', 'value': -65.8895263671875}}},
   '*2': {'*0': {'major': {'unit': 'arcsec', 'value': 0.652698278427124},
     'minor': {'unit': 'arcsec', 'value': 0.50435870885849},
     'positionangle': {'unit': 'deg', 'value': -65.88955688476562}}},
   '*3': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526971459388733},
     'minor': {'unit': 'arcsec', 'value': 0.5043579339981079},
     'positionangle': {'unit': 'deg', 'value': -65.88956451416016}}},
   '*4': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526962518692017},
     'minor': {'unit': 'arcsec', 'value': 0.5043571591377258},
     'positionangle': {'unit': 'deg', 'value': -65.88956451416016}}},
   '*5': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526952385902405},
     'minor': {'unit': 'arcsec', 'value': 0.5043563842773438},
     'positionangle': {'unit': 'deg', 'value': -65.88957977294922}}},
   '*6': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526942849159241},
     'minor': {'unit': 'arcsec', 'value': 0.5043556690216064},
     'positionangle': {'unit': 'deg', 'value': -65.88958740234375}}},
   '*7': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526930928230286},
     'minor': {'unit': 'arcsec', 'value': 0.5043548941612244},
     'positionangle': {'unit': 'deg', 'value': -65.88959503173828}}},
   '*8': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526919603347778},
     'minor': {'unit': 'arcsec', 'value': 0.5043540000915527},
     'positionangle': {'unit': 'deg', 'value': -65.88961029052734}}},
   '*9': {'*0': {'major': {'unit': 'arcsec', 'value': 0.6526909470558167},
     'minor': {'unit': 'arcsec', 'value': 0.5043532848358154},
     'positionangle': {'unit': 'deg', 'value': -65.88957214355469}}}},
  'nChannels': 10,
  'nStokes': 1},
 'refpix': array([125., 125.,   0.,   0.]),
 'refval': array([ 2.88792330e+00, -6.05713443e-01,  1.00000000e+00,  3.
↪72520023e+11]),
 'shape': array([250, 250,   1,  10]),
 'tileshape': array([125,  50,   1,   5]),
 'unit': 'Jy/beam'}
```

[36]:
```
# CNGI
print(image_xds)
```

```
<xarray.Dataset>
Dimensions:          (chan: 10, l: 250, m: 250, pol: 1, time: 1)
Coordinates:
  * chan             (chan) float64 3.725e+11 3.725e+11 ... 3.725e+11 3.725e+11
    declination      (l, m) float64 dask.array<chunksize=(250, 250), meta=np.ndarray>
  * l                (l) float64 6.06e-05 6.012e-05 ... -5.963e-05 -6.012e-05
  * m                (m) float64 -6.06e-05 -6.012e-05 ... 5.963e-05 6.012e-05
  * pol              (pol) float64 1.0
    right_ascension  (l, m) float64 dask.array<chunksize=(250, 250), meta=np.ndarray>
  * time             (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    AUTOMASK         (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,␣
↪1, 1, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    IMAGE            (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,␣
↪1, 1, 1), meta=np.ndarray>
    IMAGE_MASK0      (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,␣
↪1, 1), meta=np.ndarray>
    MODEL            (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,␣
↪1, 1, 1), meta=np.ndarray>
    PB               (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,␣
↪1, 1, 1), meta=np.ndarray>
    PB_MASK0         (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,␣
↪1, 1), meta=np.ndarray>
    PSF              (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,␣
↪1, 1, 1), meta=np.ndarray>
    RESIDUAL         (l, m, time, chan, pol) float64 dask.array<chunksize=(250, 250,␣
↪1, 1, 1), meta=np.ndarray>
    RESIDUAL_MASK0   (l, m, time, chan, pol) bool dask.array<chunksize=(250, 250, 1,␣
↪1, 1), meta=np.ndarray>
    SUMWT            (time, chan, pol) float64 dask.array<chunksize=(1, 1, 1),␣
↪meta=np.ndarray>
Attributes: (12/19)
    axisnames:            ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:            ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:           [0.6526992917060852, 0.5043594837188721, -65.889526...
    commonbeam_units:     ['arcsec', 'arcsec', 'deg']
    date_observation:     2012/11/19/07
    direction_reference:  j2000
    ...                   ...
    restoringbeam:        [0.6526992917060852, 0.5043594837188721, -65.889526...
    spectral__reference:  lsrk
    telescope:            alma
    telescope_position:   [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                 Jy/beam
    velocity__type:       radio
```

[37]:
```
# Delta
print('max delta : ', np.abs(casa_dict['incr'] - image_xds.incr).max())
```

```
max delta :  0.0
```

10.2.2 imval

Note the mechanism for retrieving data from the image is fundamentally different, but the data itself is the same

[38]:
```
# CASA6
from casatasks import imval

casa_dict = imval('twhya.image', box='85,100,86,102', chans='2')
casa_dict
```

[38]:
```
{'axes': [[0, 'Right Ascension'],
  [1, 'Declination'],
  [3, 'Frequency'],
  [2, 'Stokes']],
 'blc': [85, 100, 0, 2],
 'coords': array([[[ 2.88794689e+00, -6.05725563e-01,  1.00000000e+00,
```

(continues on next page)

(continued from previous page)

```
           3.72521243e+11],
         [ 2.88794689e+00, -6.05725079e-01,  1.00000000e+00,
           3.72521243e+11],
         [ 2.88794689e+00, -6.05724594e-01,  1.00000000e+00,
           3.72521243e+11]],

        [[ 2.88794630e+00, -6.05725563e-01,  1.00000000e+00,
           3.72521243e+11],
         [ 2.88794630e+00, -6.05725079e-01,  1.00000000e+00,
           3.72521243e+11],
         [ 2.88794630e+00, -6.05724594e-01,  1.00000000e+00,
           3.72521243e+11]]]),
 'data': array([[[0.01665862, 0.02118381, 0.02057032],
        [0.00525612, 0.00775176, 0.00969939]]]),
 'mask': array([[[ True,  True,  True],
        [ True,  True,  True]]]),
 'trc': [86, 102, 0, 2],
 'unit': 'Jy/beam'}
```

[39]:
```
# CNGI
cngi_xds = image_xds.isel(l=range(85,87), m=range(100,103), chan=2).compute()
print(cngi_xds)
```

```
<xarray.Dataset>
Dimensions:           (l: 2, m: 3, pol: 1, time: 1)
Coordinates:
    chan              float64 3.725e+11
    declination       (l, m) float64 -0.6057 -0.6057 -0.6057 ... -0.6057 -0.6057
  * l                 (l) float64 1.939e-05 1.891e-05
  * m                 (m) float64 -1.212e-05 -1.164e-05 -1.115e-05
  * pol               (pol) float64 1.0
    right_ascension   (l, m) float64 2.888 2.888 2.888 2.888 2.888 2.888
  * time              (time) datetime64[ns] 2012-11-19T07:56:26.544000626
Data variables:
    AUTOMASK          (l, m, time, pol) float64 0.0 0.0 0.0 0.0 0.0 0.0
    IMAGE             (l, m, time, pol) float64 0.01666 0.02118 ... 0.009699
    IMAGE_MASK0       (l, m, time, pol) bool True True True True True True
    MODEL             (l, m, time, pol) float64 0.0 0.0 0.0 0.0 0.0 0.0
    PB                (l, m, time, pol) float64 0.8004 0.8046 ... 0.8115 0.8156
    PB_MASK0          (l, m, time, pol) bool True True True True True True
    PSF               (l, m, time, pol) float64 -0.01334 -0.01428 ... 0.0001316
    RESIDUAL          (l, m, time, pol) float64 0.01666 0.02118 ... 0.009699
    RESIDUAL_MASK0    (l, m, time, pol) bool True True True True True True
    SUMWT             (time, pol) float64 1.13e+07
Attributes: (12/19)
    axisnames:             ['Right Ascension', 'Declination', 'Time', 'Frequen...
    axisunits:             ['rad', 'rad', 'datetime64[ns]', 'Hz', '']
    commonbeam:            [0.6526992917060852, 0.5043594837188721, -65.889526...
    commonbeam_units:      ['arcsec', 'arcsec', 'deg']
    date_observation:      2012/11/19/07
    direction_reference:   j2000
    ...                    ...
    restoringbeam:         [0.6526992917060852, 0.5043594837188721, -65.889526...
    spectral__reference:   lsrk
    telescope:             alma
    telescope_position:    [2.22514e+06m, -5.44031e+06m, -2.48103e+06m] (itrf)
    unit:                  Jy/beam
```

(continues on next page)

(continued from previous page)

```
    velocity__type:       radio
```

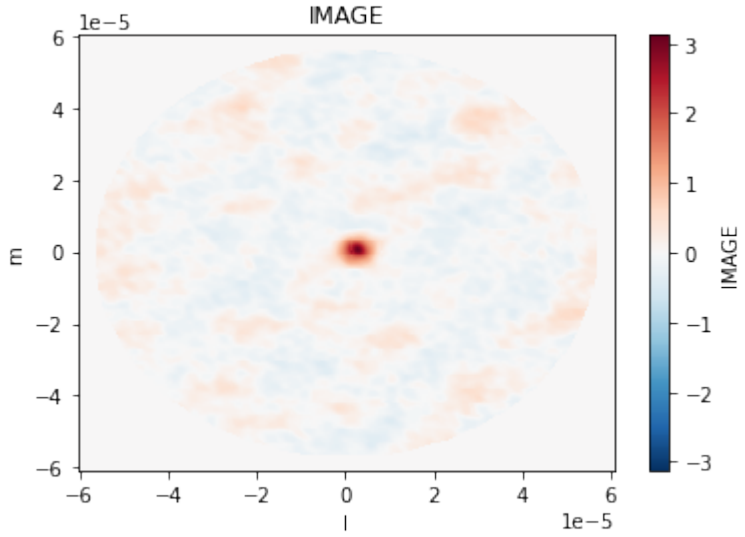
[40]:
```
# Delta
print('max delta : ', np.abs(casa_dict['data'] - cngi_xds.IMAGE.values.squeeze()).
↪max())
```

```
max delta :  0.0
```

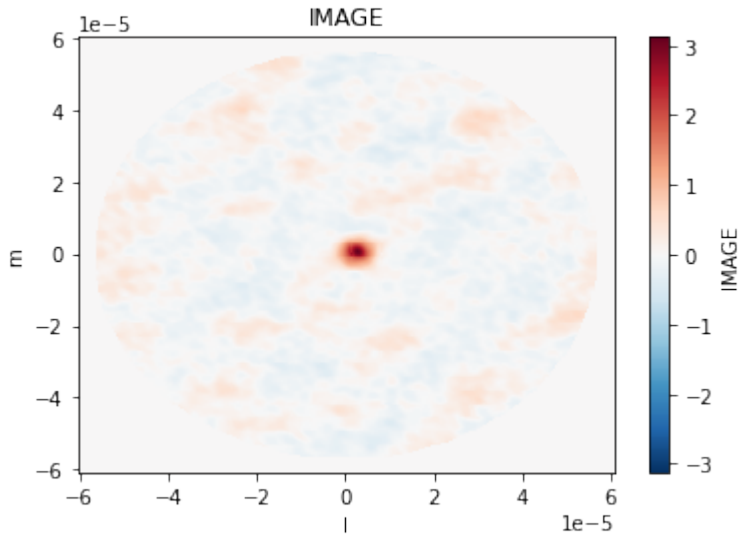
10.2.3 imcollapse

[41]:
```
# CASA6
from casatasks import imcollapse

imcollapse('twhya.image', function='sum', axes=[2,3], outfile='casa6.collapse.image',␣
↪overwrite=True)

casa_xds = convert_image('casa6.collapse.image')
implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.21397114 seconds
```



[42]:
```
# CNGI
cngi_xds = image_xds.where(image_xds.IMAGE_MASK0).sum(dim=['chan','pol'])

implot(cngi_xds.IMAGE)
```



[43]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xds.IMAGE).values).max())
```

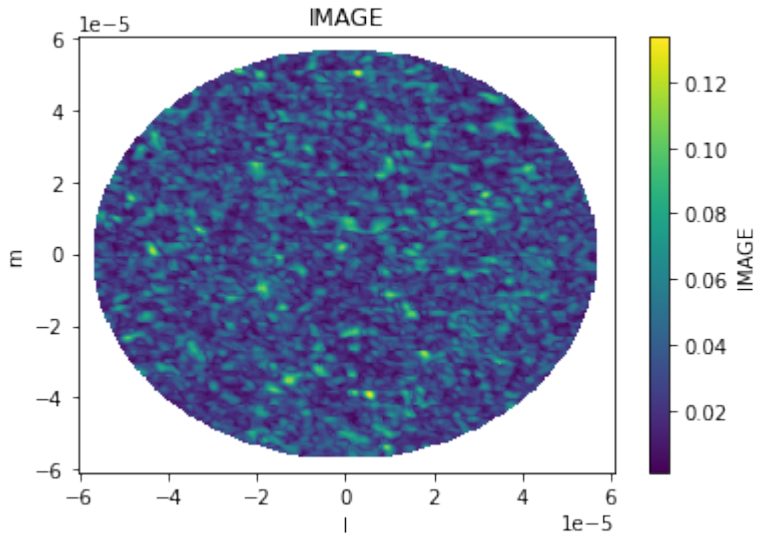
```
max delta :  1.1920928955078125e-07
```

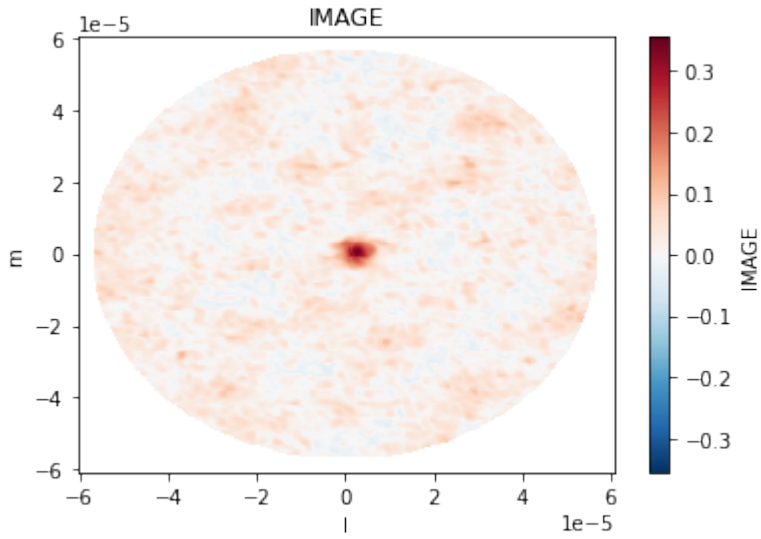
10.2.4 imcontsub

Apparently someone the naming of line and continuum files is reversed

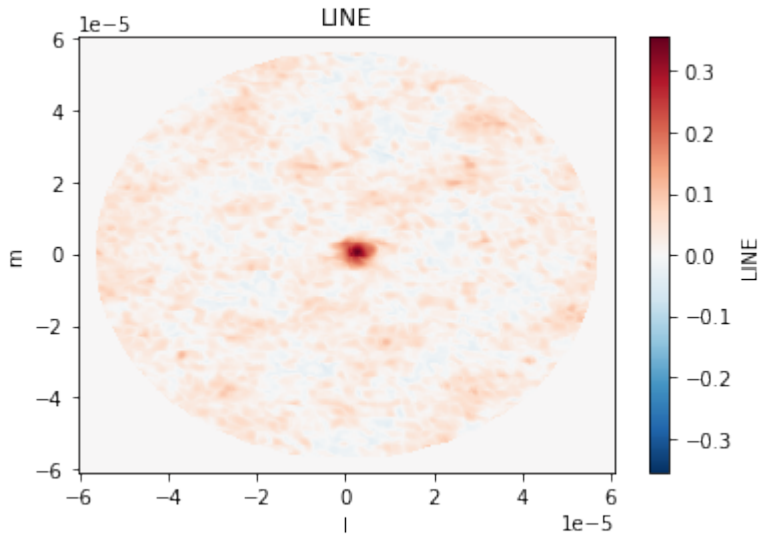
[44]:
```
# CASA6
from casatasks import imcontsub

os.system('rm -fr casa6.*.image')
imcontsub(imagename="twhya.image", linefile="casa6.line.image", contfile="casa6.cont.
↪image", fitorder=2, chans='1~3,7~9')

casa_linefit = convert_image('casa6.line.image')
casa_contsub = convert_image('casa6.cont.image')

implot(casa_linefit.IMAGE.where(casa_linefit.IMAGE_MASK0))
implot(casa_contsub.IMAGE.where(casa_contsub.IMAGE_MASK0))
```

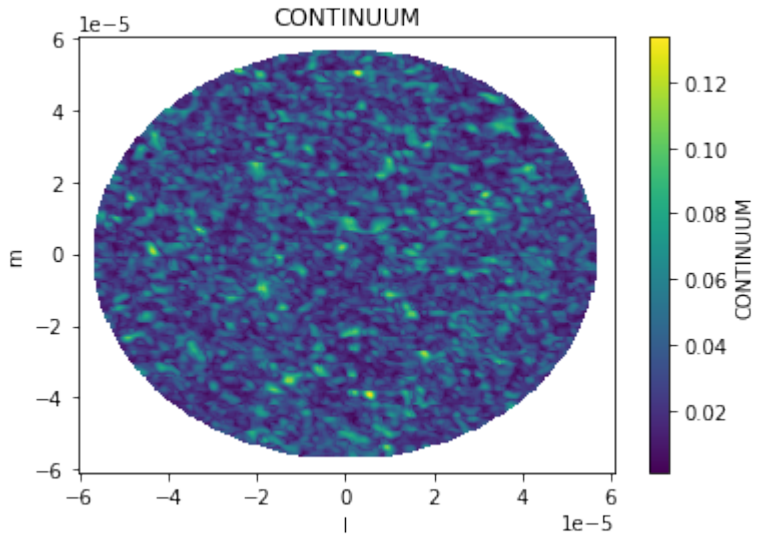
```
converting Image...
processed image in 0.38849592 seconds
converting Image...
processed image in 0.37773752 seconds
```





[45]:
```
# CNGI
from cngi.image import cont_sub

cngi_xds = cont_sub(image_xds.where(image_xds.IMAGE_MASK0), dv='IMAGE', fitorder=2,␣
↪chans=[1,2,3,7,8,9], linename='LINE', contname='CONTINUUM')

implot(cngi_xds.LINE)
implot(cngi_xds.CONTINUUM)
```





[46]:
```
# Delta
print('max delta : ', np.nanmax(np.abs((casa_linefit.IMAGE - cngi_xds.CONTINUUM).
↪values)))
print('max delta : ', np.nanmax(np.abs((casa_contsub.IMAGE.where(casa_contsub.IMAGE_
↪MASK0) - cngi_xds.LINE).values)))
```

```
max delta :  1.4396755787515758e-08
max delta :  1.4896446964840493e-08
```

10.2.5 imdev - TBD

Pending implementation of statistics, this will likely be done directly on the xds by calling xds.rolling() with the new statistics function

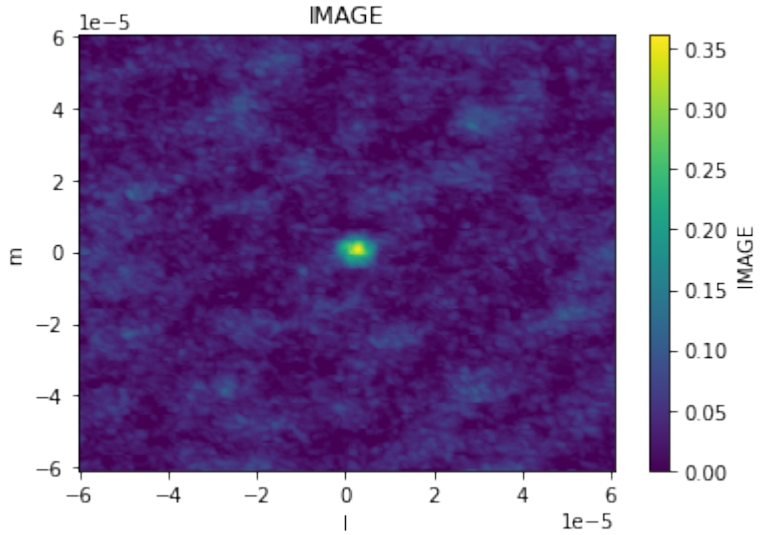
---

10.2.6 immath

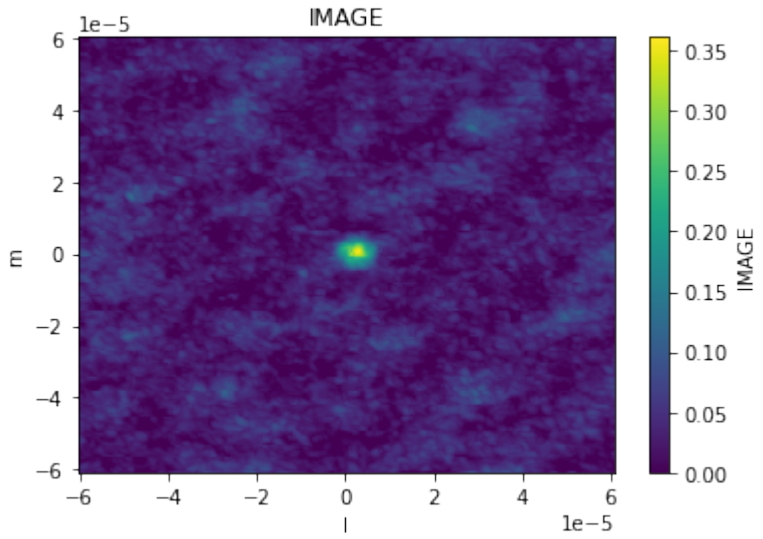
Note that only 'evalexpr' mode is shown here. 'spix' mode should instead use cngi.image.spixfit and 'poli'/'pola' modes should compute the math manually

[47]:
```
# CASA6
from casatasks import immath

os.system("rm -fr casa6.math.image")
immath(imagename='twhya.image', expr='sin(IM0)', outfile='casa6.math.image')

casa_xds = convert_image('casa6.math.image')
implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.39214063 seconds
```



[48]:
```
# CNGI
cngi_xda = np.sin(image_xds.IMAGE)

implot(cngi_xda)
```



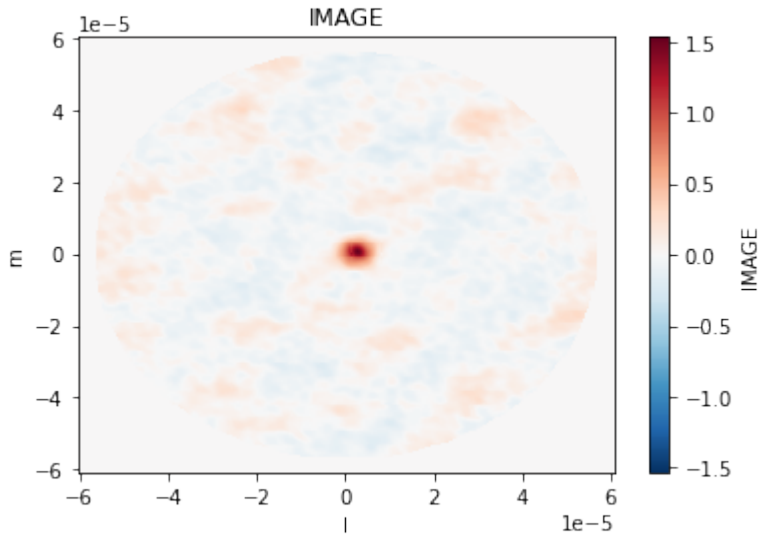
[49]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xda).values).max())
```

```
max delta :  1.4870845721492998e-08
```

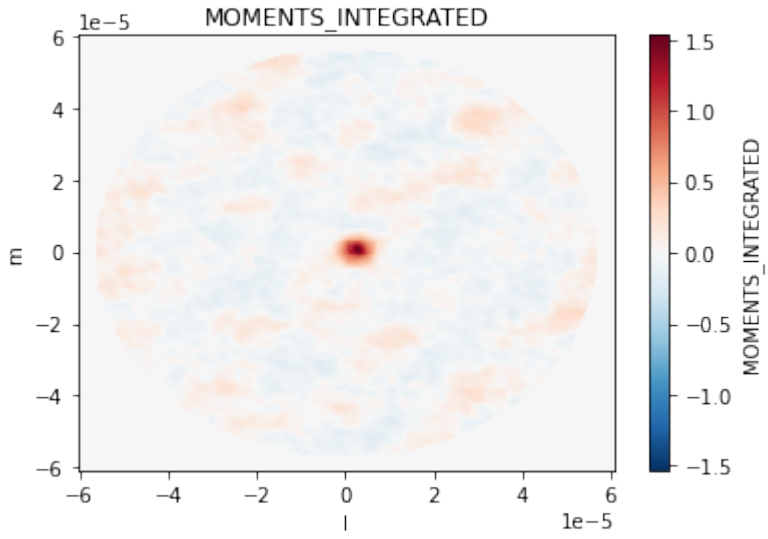
10.2.7 immoments

[50]:
```
# CASA6
from casatasks import immoments

os.system('rm -rf casa6.immoments.image*')
immoments(imagename='twhya.image', axis='spectral', moments=[0], outfile='casa6.
↪immoments.image')

casa_xds = convert_image('casa6.immoments.image')

implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.20693302 seconds
```



[51]:
```
# CNGI
from cngi.image import moments

cngi_xds = moments(image_xds.where(image_xds.IMAGE_MASK0), moment=[0])

implot(cngi_xds.MOMENTS_INTEGRATED)
```



[52]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xds.MOMENTS_INTEGRATED).values).
↪max())
```

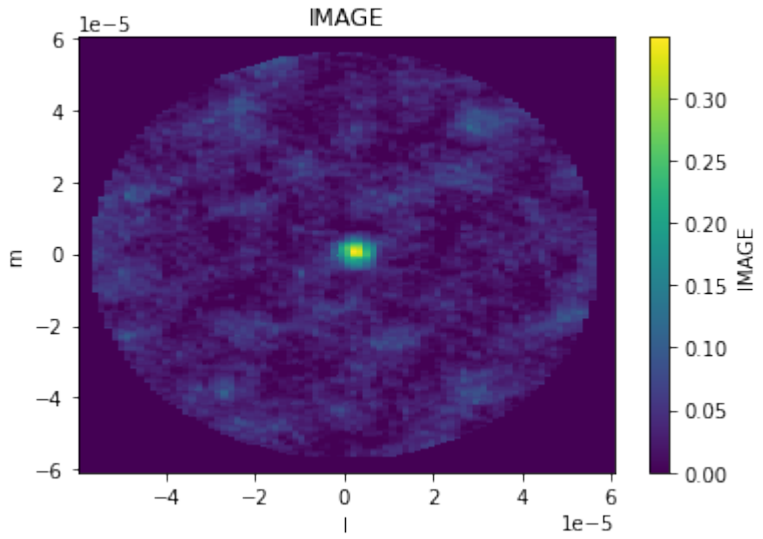
```
max delta :  5.094286832374451e-07
```

10.2.8 imrebin

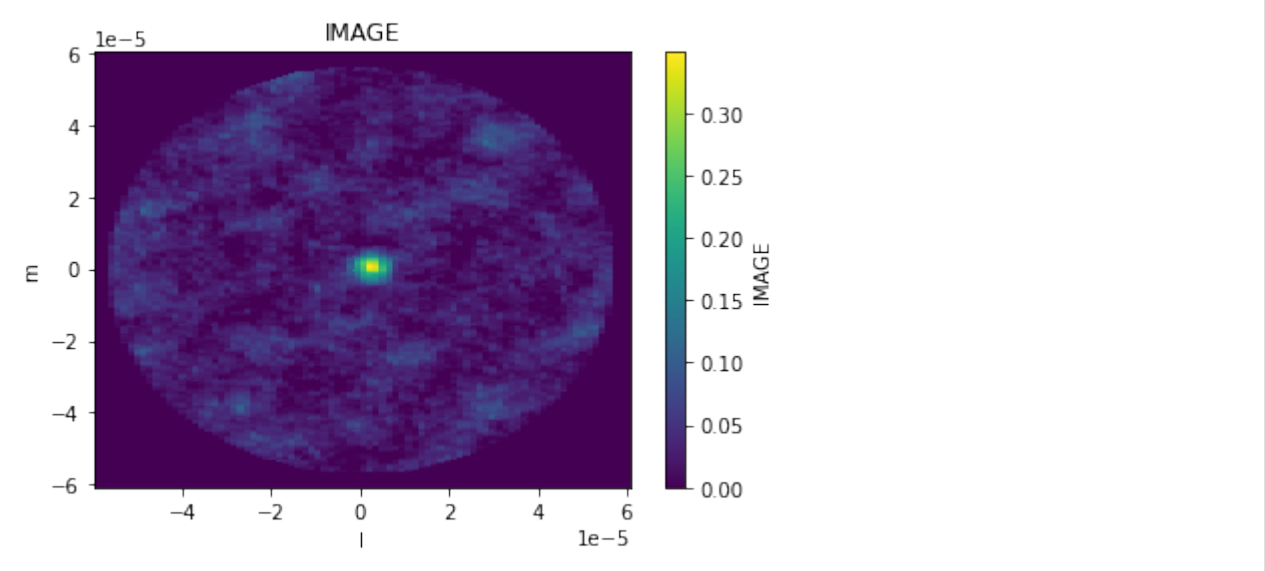
Note that CASA bins pixels starting from the edge of the mask/region. CNGI bins pixels starting from the edge of the image (ignoring values in masked pixels). This creates a discrepency in output at the edge of masks.

[53]:
```
# CASA6
from casatasks import imrebin

imrebin(imagename='twhya.image', outfile='casa6.rebin.image', factor=3,␣
↪overwrite=True)

casa_xds = convert_image('casa6.rebin.image')
implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.2709217 seconds
```



[54]:
```
# CNGI
from cngi.image import rebin

cngi_xds = rebin(image_xds.where(image_xds.IMAGE_MASK0), axis='l', factor=3)

implot(cngi_xds.IMAGE)
```



[55]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xds.IMAGE).values).max())
```

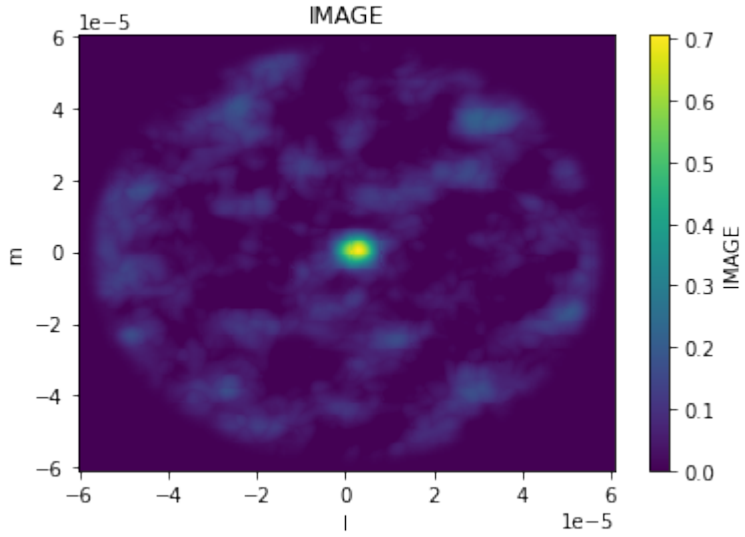
```
max delta :  0.05339271823565166
```

10.2.9 imsmooth

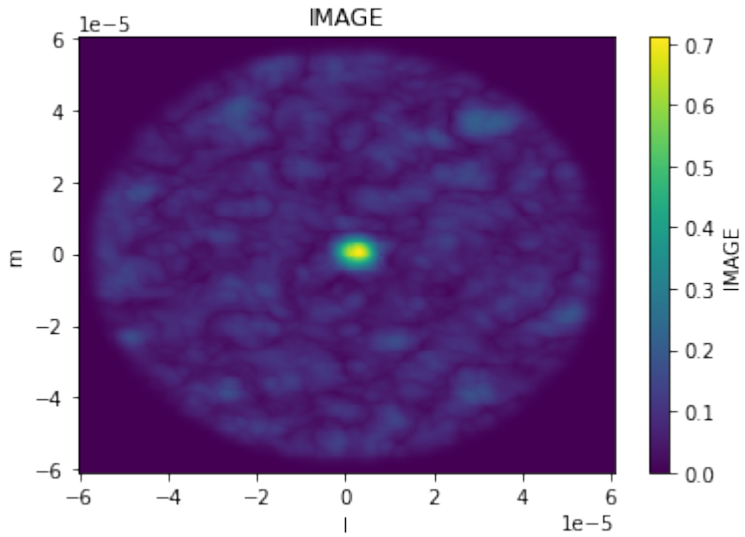
Note there is currently a discrepency in output that is not understood

[56]:
```
# CASA6
from casatasks import imsmooth

imsmooth('twhya.image', kernel='gaussian', targetres=True, outfile='casa6.smooth.image
↪',
         overwrite=True, beam = {"major": "1arcsec", "minor": "1arcsec", "pa": "30deg
↪"});

casa_xds = convert_image('casa6.smooth.image')
implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.38987756 seconds
```



[57]:
```
# CNGI
from cngi.image import smooth

cngi_xds = smooth(image_xds.where(image_xds.IMAGE_MASK0, other=0), kernel='gaussian',␣
↪size=[1., 1., 30.],
                  current=image_xds.commonbeam, name='TARGET_BEAM')

implot(cngi_xds.IMAGE)
```



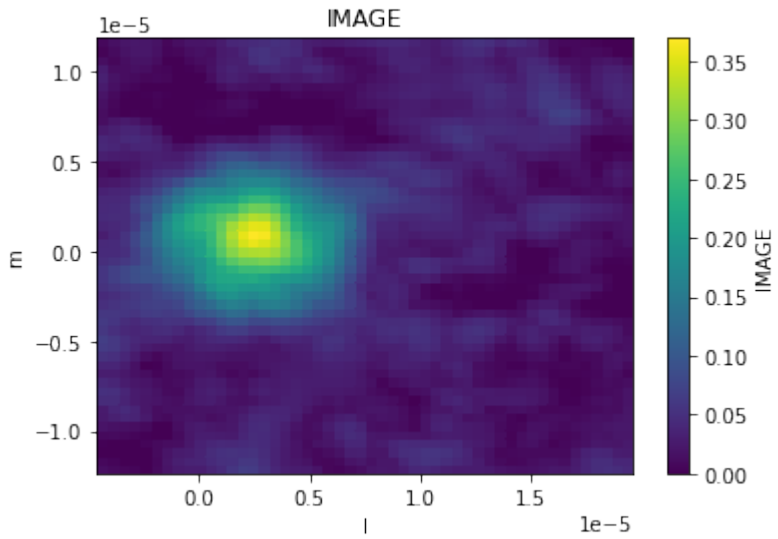
[58]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xds.IMAGE).values).max())
```

```
max delta :  0.3243054985856718
```

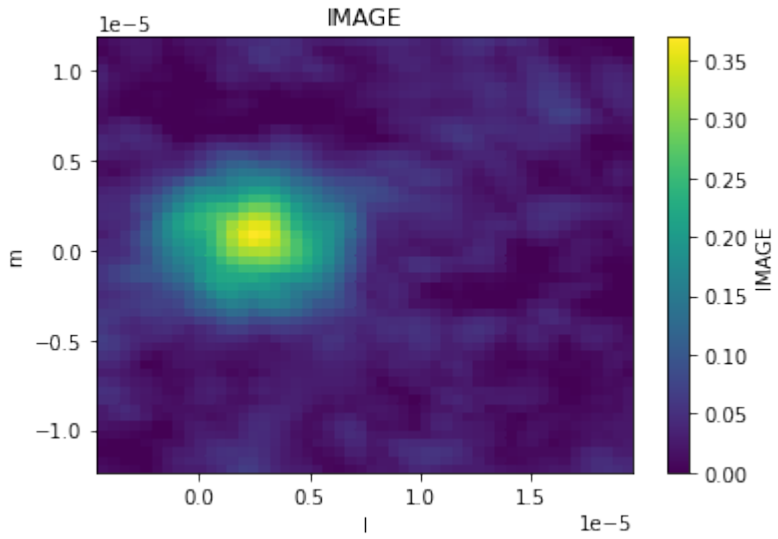
10.2.10 imsubimage

[59]:
```
# CASA6
from casatasks import imsubimage

imsubimage('twhya.image', outfile='casa6.subimage.image', box='85,100,134,149',␣
↪overwrite=True)

casa_xds = convert_image('casa6.subimage.image')
implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.1738503 seconds
```



[60]:
```
# CNGI
cngi_xds = image_xds.isel(l=range(85,135), m=range(100,150))

implot(cngi_xds.IMAGE)
```



[61]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xds.IMAGE).values).max())
```

```
max delta :  0.0
```

10.2.11 imstat

[62]:
```
# CASA6
from casatasks import imstat

casa_stats = imstat('twhya.image')

casa_stats
```

[62]:
```
{'blc': array([0, 0, 0, 0]),
 'blcf': '11:01:52.810, -34.42.29.866, I, 3.7252e+11Hz',
 'flux': array([6.27340375]),
 'max': array([0.36960047]),
 'maxpos': array([120, 127,   0,   1]),
 'maxposf': '11:01:51.837, -34.42.17.166, I, 3.725206e+11Hz',
 'mean': array([0.00110953]),
 'medabsdevmed': array([0.01520567]),
 'median': array([0.]),
 'min': array([-0.10551776]),
 'minpos': array([ 94, 193,   0,   3]),
 'minposf': '11:01:52.047, -34.42.10.566, I, 3.725219e+11Hz',
 'npts': array([429530.]),
 'q1': array([-0.01529752]),
 'q3': array([0.01509237]),
 'quartile': array([0.03038988]),
 'rms': array([0.02771565]),
 'sigma': array([0.02769347]),
 'sum': array([476.57430563]),
 'sumsq': array([329.94660415]),
 'trc': array([249, 249,   0,   9]),
 'trcf': '11:01:50.790, -34.42.04.966, I, 3.725255e+11Hz'}
```

[63]:
```
# CNGI
from cngi.image import statistics

cngi_stats = statistics(image_xds, compute=True).statistics

cngi_stats
```

[63]:
```
{'blc': array([0, 0, 0, 0, 0]),
 'blcf': ['11:01:52.80971154',
  '-34.42.29.86573768',
  '2012-11-19T07:56:26.544000626',
  372520022603.63745,
  1.0],
 'max': 0.36960047483444214,
 'maxpos': [120, 127, 0, 1, 0],
 'maxposf': ['11:01:51.83654673',
  '-34.42.17.16599958',
  '2012-11-19T07:56:26.544000626',
```

(continues on next page)

(continued from previous page)

```
 372520632933.7965,
 1.0],
'mean': 0.0005451506748828763,
'medabsdevmed': 0.014964754227548838,
'median': 0.0,
'min': -0.10752750933170319,
'minpos': [27, 199, 0, 3, 0],
'minposf': ['11:01:52.59069674',
 '-34.42.09.96583877',
 '2012-11-19T07:56:26.544000626',
 372521853594.1146,
 1.0],
'npts': 625000,
'q1': -0.01533513329923153,
'q3': 0.014549590414389968,
'quartile': 0.029884723713621497,
'rms': 0.026438209534621095,
'sigma': 0.026432588487286,
'sum': 340.71917180179764,
'sumsq': 436.8618271228311,
'trc': array([249, 249,   0,   9,   0]),
'trcf': ['11:01:50.79048222',
 '-34.42.04.96574187',
 '2012-11-19T07:56:26.544000626',
 372525515575.069,
 1.0]}
```

[64]:
```
# Delta
print('max delta : ', np.abs([casa_stats['medabsdevmed']-cngi_stats['medabsdevmed'],␣
↪casa_stats['q1']-cngi_stats['q1']]).max())
```

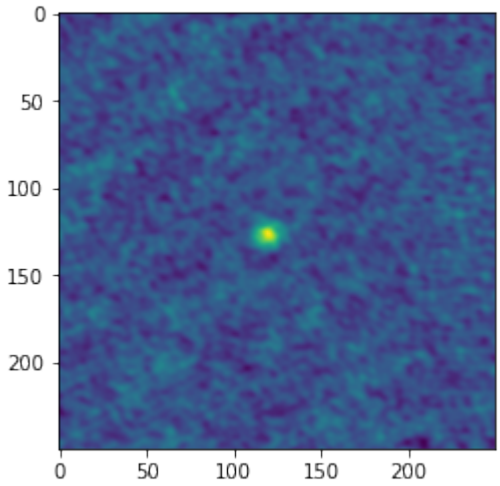
```
max delta :  0.00024092057719826698
```

10.2.12 imtrans

Note the image converter will swap the transposed axes back, so we have to work around that here to show the effect

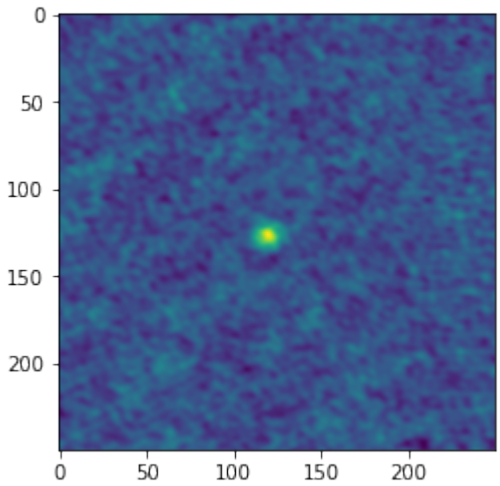
[65]:
```
# CASA6
from casatasks import imtrans

os.system("rm -fr casa6.trans.image")
imtrans('twhya.image', outfile='casa6.trans.image', order='1023')

casa_image = imval('casa6.trans.image', box='0,0,249,249', chans='2')['data']

plt.imshow(casa_image)
```

[65]:
```
<matplotlib.image.AxesImage at 0x7f5de9a22690>
```



[66]:
```
# CNGI
cngi_xds = image_xds.transpose('m','l','time','chan','pol')

plt.imshow(cngi_xds.IMAGE[:,:,0,2,0].values)
```

[66]:
```
<matplotlib.image.AxesImage at 0x7f5de7798950>
```



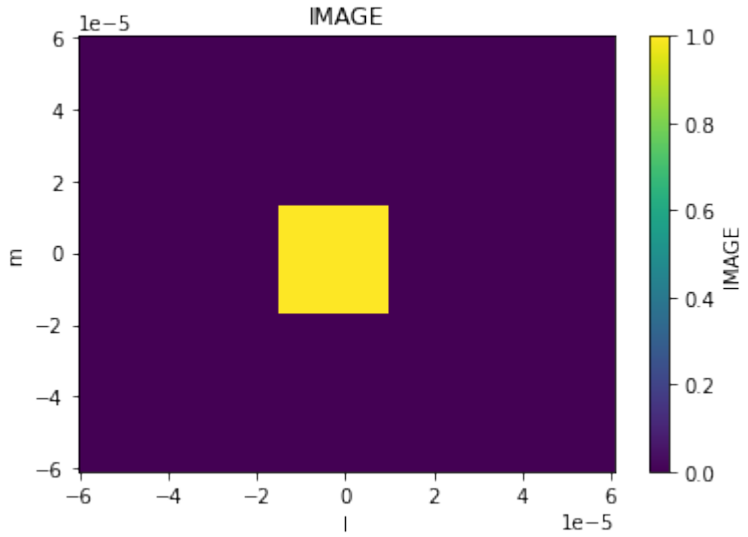
[67]:
```
# Delta
print('max delta : ', np.abs(casa_image - cngi_xds.IMAGE[:,:,0,2,0].values).max())
```

```
max delta :  0.0
```

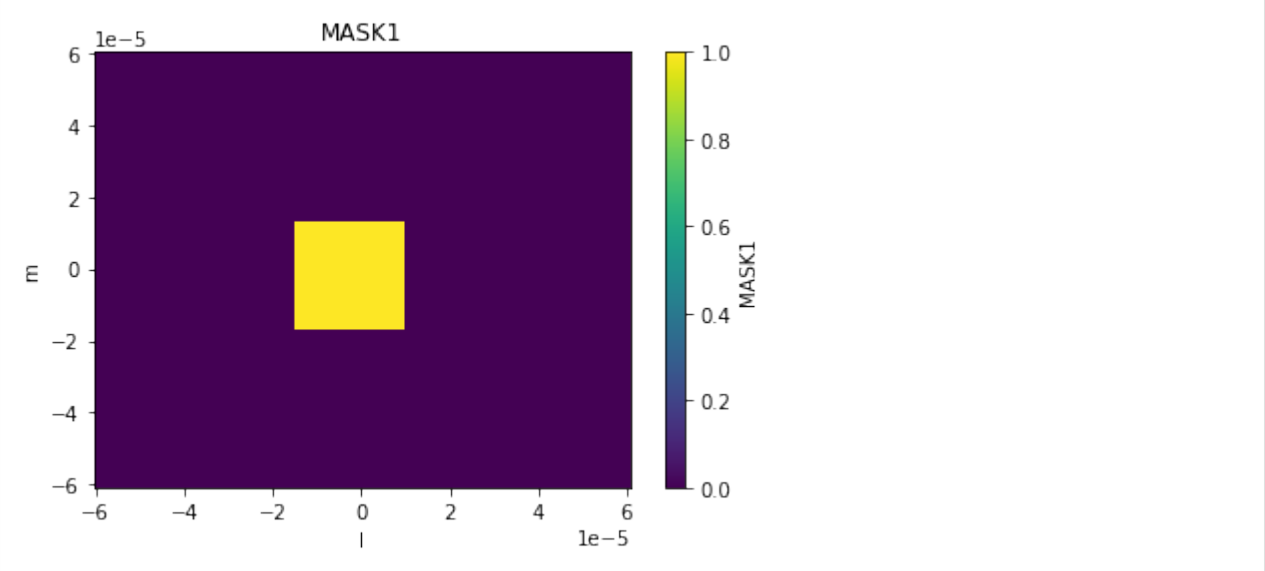
10.2.13 makemask

[68]:
```
# CASA6
from casatasks import makemask

makemask(mode='copy', inpimage='twhya.image', inpmask='box[[2.887905rad, -0.60573rad],
↪ [2.887935rad, -0.60570rad]]',
         output='casa6.makemask.image', overwrite=True)

casa_xds = convert_image('casa6.makemask.image')
implot(casa_xds.IMAGE)
```

```
converting Image...
processed image in 0.22030282 seconds
```



[69]:
```
# CNGI
from cngi.image import region

# note that CASA swaps the meaning of 0/1 between regions and masks, CNGI does not
# so replicating the values of a CASA mask requires us to use a CNGI region
cngi_xds = region(image_xds, 'MASK1', ra=[2.887905, 2.887935], dec=[-0.60573, -0.
↪60570])

implot(cngi_xds.MASK1)
```



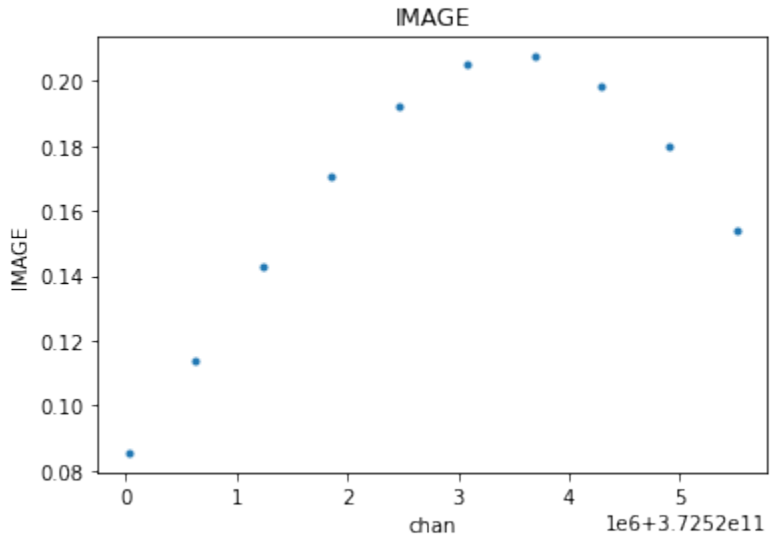
[70]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE - cngi_xds.MASK1).values).max())
```

```
max delta :  0.0
```

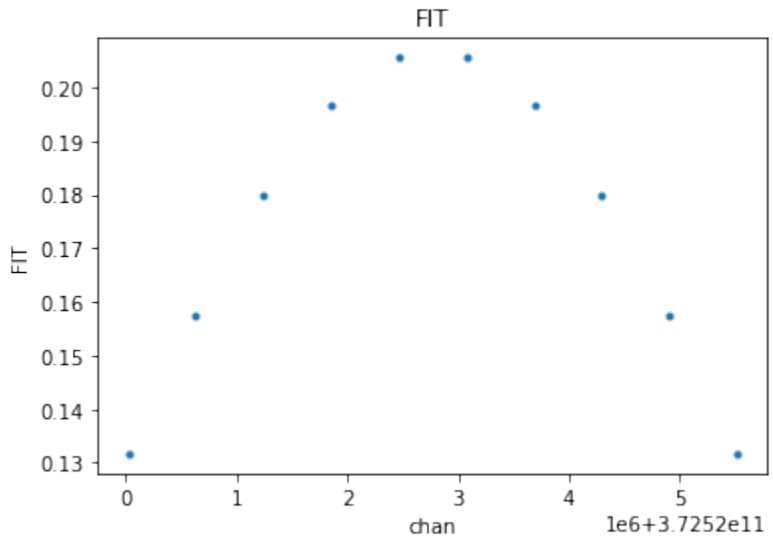
10.2.14 specfit

[71]:
```
from casatasks import specfit

os.system('rm -fr casa6.specfit.image')
rc = specfit('twhya.image', box="125,125,125,125", model='casa6.specfit.image',␣
↪pampest=1.5, pcenterest=0, pfwhmest=5, ngauss=1, multifit=False)

casa_xds = convert_image('casa6.specfit.image')
implot(casa_xds.IMAGE[0,0,0,:,0], axis=['chan'])
```

```
converting Image...
processed image in 0.16069746 seconds
```



[72]:
```
from cngi.image import spec_fit

cngi_xds = spec_fit(image_xds, dv='IMAGE', pixel=(125,125), sigma=2000, name='FIT')

implot(cngi_xds.FIT, axis='chan', overplot=True)
```



[73]:
```
# Delta
print('max delta : ', np.abs((casa_xds.IMAGE[0,0,0,:,0] - cngi_xds.FIT).values).max())
```

```
max delta :  0.04620384288038856
```

Open in Colab: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/development.ipynb

CHAPTER

ELEVEN

DEVELOPMENT

READ THIS BEFORE YOU CONTRIBUTE CODE!!!

11.1 Organization

The CNGI Prototype is organized into *packages* and *modules* as described below. Base infrastructure, data manipulation and mathematics are contained in the CNGI package. Higher level data reduction functionality is contained in the ngCASA package. Within each package are a set of modules, with each module responsible for a different functional area.

CNGI package

- **conversion** : convert legacy CASA data files to CNGI compatible files
- **dio** : data objects to/from CNGI compatible data files
- **vis** : operations on visibility data objects
- **image** : operations on image data objects
- **direct** : access/initialize the underlying parallel processing framework

ngCASA package

- **flagging** : Generates flags for visibility data.
- **calibration** : Generates and applies calibration solutions to visibility data.
- **imaging** : Converts visibility data to images and applies antenna primary beam and w-term corrections.
- **deconvolution** : Deconvolves PSF from images and combines images.

The cngi_prototype repository on GitHub contains both packages along with supporting folders `docs` and `tests`.

11.2 Architecture

The CNGI Prototype application programming interface (API) is a set of flat, stateless functions that take an xarray Dataset as an input parameter and return a new xarray Dataset (XDS) as output. The term **"flat"** means that the functions are not allowed to call each other, and the term **"stateless"** means that they may not access any global data outside the parameter list, nor maintain any persistent internal data.

The CNGI Prototype code base is not object oriented, and instead follows a more functional paradigm. Objects are indeed used to hold Visibility and Image data, but they come directly from the underlying xarray/dask framework and are not extended in any way.

The data variables in the XDS are associated with task graphs. Compute is only triggered when the user explicitly calls compute on the XDS or instructs a function to save to disk.

The CNGI Prototype adheres to a strict design philosophy with the following **RULES**:

1. Each file in a module must have exactly one function exposed to the external API (by docstring and __init__.py).
2. The exposed function name should match the file name.
3. Must use stateless functions, not classes.
4. Files in a module cannot import each other.
5. Files in separate modules cannot import each other.
6. Special _utility modules may exist for internal functions meant to be shared across modules/files. But each module file should be as self contained as possible.
7. A module's utility files may only be imported by that module's API files.
8. A package's utility files may only be imported by other modules in that package.
9. No utility functions may be exposed to the external API.

```
cngi_prototype
|-- cngi
|     |-- module1
|     |      |-- __init__.py
|     |      |-- file1.py
|     |      |-- file2.py
|     |      | ...
|     |-- module2
|     |      |-- __init__.py
|     |      |-- file3.py
|     |      |-- file4.py
|     |      | ...
|     |-- _utils
|     |      |-- __init__.py
|     |      |-- _file5.py
|     |      |-- _file6.py
|     |      | ...
|     | ...
|-- ngcasa
|     |-- module1
|     |      |-- __init__.py
|     |      |-- file1.py
|     |      |-- file2.py
|     |      | ...
|     |      |-- _module1_utils
|     |      |      |-- __init__.py
```

(continues on next page)

(continued from previous page)

```
|     |      |      |-- _check_module1_parms.py
|     |      |      |-- _file5.py
|     |      |      |-- _file6.py
|     |      |      | ...
|     |-- module2
|     |      |-- __init__.py
|     |      |-- file3.py
|     |      |-- file4.py
|     |      | ...
|     |      |-- _module2_utils
|     |      |      |-- __init__.py
|     |      |      |-- _check_module2_parms.py
|     |      |      |-- _file7.py
|     |      |      |-- _file8.py
|     |      |      | ...
|     | ...
|-- docs
|     | ...
|-- tests
|     | ...
|-- requirements.txt
|-- setup.py
```

File1, file2, file3 and file4 **MUST** be documented in the API exactly as they appear. They must **NOT** import each other. _file5 and _file6 are utility files, they must **NOT** be documented in the API. They may be imported by file1-4 (cngi package) and file1-2 (ngcasa package).

There are several important files to be aware of:

- **__init__.py** : dictates what is seen by the API and importable by other functions
- **requirements.txt** : lists all library dependencies for development, used by IDE during setup
- **setup.py** : defines how to package the code for pip, including version number and library dependencies for installation
- **_check_module_parms.py** : Each module has a _check_module_parms.py file that has functions that check the input parameters of the module's API functions. The parameter defaults are also defined here.
- **_check_parms.py** : Provides the _check_parms and _check_storage_parms functions. The _check_parms is used by all the _check_module_parms.py files to check parameter data types, values and set defaults. The storage_parm is parameter that is common to all API functions is checked by _check_storage_parms.
- **_store.py** : Provides the _store function that stores datasets or appends data variables. All API functions use this function.

11.2.1 CNGI Function Template

```
def myfunction(xds, param_1, ..., param_m):
    """
    Description of function

    Parameters
    ----------
    xds : xarray.core.dataset.Dataset
```

(continues on next page)

(continued from previous page)

```
    input xarray dataset
param_1 : type
    description of this parameter
...
parms_n : type
    description of this parameter

Returns
-------
xarray.core.dataset.Dataset
    new output xarray dataset
"""

### Import Statements
import numpy as np
...

### Parameter Checking
assert(param_1 == what_it_shoud, "ERROR: param_1 is wrong")
...

### Function code
c = xds.a + xds.b
...

### Return a new xds, leaving input untouched
nxds = xds.assign({'mynewthing':c}

return nxds
```

By default calling an CNGI function will not do computation, but rather build a graph of the computation (example of a graph). Computation can be triggered by using `dask.compute(dataset)`.

11.2.2 Data Structures

Data is stored in Zarr files with Xarray formatting (which is specified in the metadata). The data is stored as N-dimensional arrays that can be chunked on any axis. The ngCASA/CNGI data formats are:

- vis.zarr Visibility data (measurement set).
- img.zarr Images (also used for convolution function cache).
- cal.zarr Calibration tables.
- tel.zar Telescope layout (used for simulations)
- Other formats will be added as needed.

CNGI will provide functions to convert between the new Zarr formats and legacy formats (such as the measurement set, FITS files, ASDM, etc.). The current implementations of the Zarr formats are not fully developed and are only sufficient for prototype development.

Zarr data formats rules:

- Data variable names are always uppercase and dimension coordinates are lowercase.
- Data variable names are not fixed but defaults exist, for example the data variable that contains the uvw data default name is UVW. This flexibility allows for the easy inclusion of more advanced algorithms (for example multi-term deconvolution that produces Taylor term images).

- Dimension coordinates and coordinates have fixed names.
- Dimension coordinates are not chunked.
- Data variables and coordinates are chunked and should have consistent chunking with each other.
- Any number of data variables can be in a dataset but must share a common coordinate and chunking system.

11.3 Framework

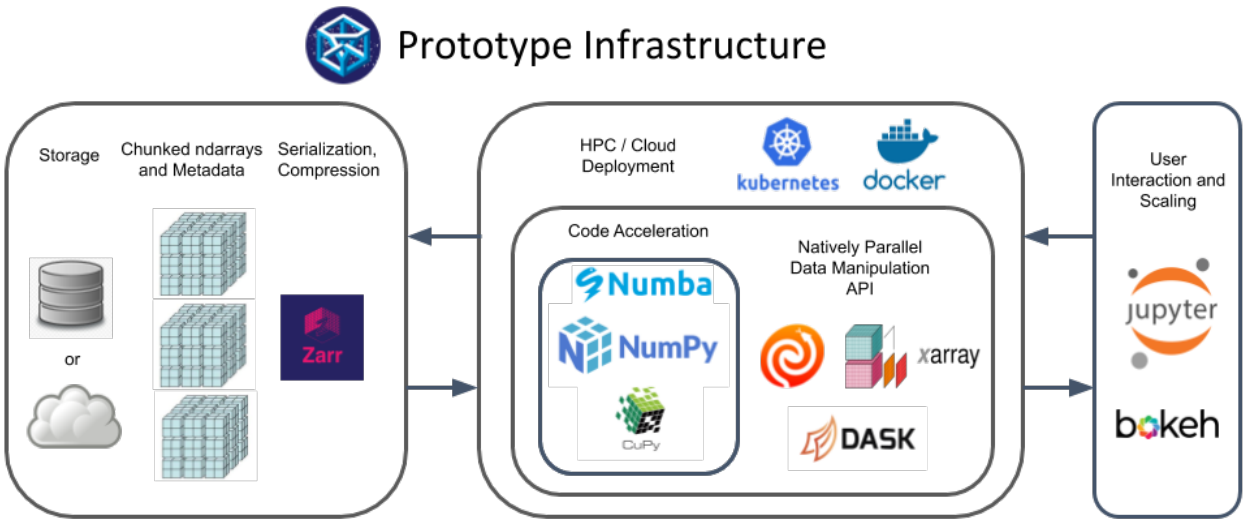
The abstraction layers of ngCASA and CNGI are unified by the paradigm of functional programming:

Build a directed acyclic graph (DAG) of blocked algorithms composed from functions (the edges) and data (the vertices) for lazy evaluation by a scheduler process coordinating a network of (optionally distributed and heterogeneous) machine resources.

This approach relies on three compatible packages to form the core of the framework:

- Dask's specificiation of the DAG coordinates processing and network resources
- Zarr's specification of blocked data structures coordinates input/output and serialization resources
- Xarray's specficiation of multidimensional indexed arrays serves as the in-memory representation and interface to the other components of the framework

The relationship between these libraries can be conceptualized by the following diagram:



In the framework data is stored in the Zarr format in a file mounted on a local disk or in the cloud. The Dask scheduler manages *N* worker processes identified to the central scheduler by their unique network address and coordinating *M* threads. Each thread applies functions to a set of data chunks. Xarray wraps Dask functions and labels the Dask arrays for ease of interface. Any code that is wrapped with Python can be parallelized with Dask. Therefore, the option to use C++, Numba or other custom high performance computing (HPC) code is retained. The size of the Zarr data chunks (on disk) and Dask data chunks do not have to be the same, this is further explained in the chunking section. Data chunks can either be read from disk as they are needed for computation or the data can be persisted into memory.

11.3.1 Dask and Dask Distributed

Dask is a flexible library for parallel computing in Python and Dask Distributed provides a centrally managed, distributed, dynamic task scheduler. A Dask task graph describes how tasks will be executed in parallel. The nodes in a task graph are made of Dask collections. Different Dask collections can be used in the same task graph. For the CNGI and ngCASA projects Dask `array` and Dask `delayed` collections will predominantly be used. Explanations from the Dask website about Dask arrays and Dask delayed:

> Dask array implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays. This lets us compute on arrays larger than memory using all of our cores. We coordinate these blocked algorithms using Dask graphs.
>
> Sometimes problems don't fit into one of the collections like dask.array or dask.dataframe. In these cases, users can parallelize custom algorithms using the simpler dask.delayed interface. This allows one to create graphs directly with a light annotation of normal python code.

The following diagram taken from the Dask website illustrates the components of the chosen parallelism framework.

Dask/Dask Distributed Advantages:

- Parallelism can be achieved over any dimension, since it is determined by the data chunking.
- Data can either be persisted into memory or read from disk as needed. As processing is finished chunks can be saved. This enables the processing of data that is larger than memory.
- Graphs can easily be combined with `dask.compute()` to concurrently execute multiple functions in parallel. For example a cube and continuum image can be created in parallel.

11.3.2 Xarray

Xarray provides N-Dimensional labeled arrays and datasets in Python. The Xarray dataset is used to organize and label related arrays. The Xarray website gives the following definition of a dataset:

- "xarray.Dataset is Xarray's multi-dimensional equivalent of a DataFrame. It is a dict-like container of labeled arrays (DataArray objects) with aligned dimensions."

The Zarr disk format and Xarray dataset specification are compatible if the Xarray `to_zarr` and `open_zarr` functions are used. The compatibility is achieved by requiring the Zarr group to have labeling information in the metadata files and a depth of one (see the Zarr section for further explanation).

When `xarray.open_zarr(zarr_group_file)` is used the array data is not loaded to memory (there is a parameter to force this), rather the metadata is loaded and a lazy loaded Xarray dataset is created. For example a dataset that consists of three dimension coordinates (dim_), two coordinates (coord_) and three data variables (Data_) would have the following structure (this is what is displayed if a print command is used on a lazy loaded dataset):

```
<xarray.Dataset>
Dimensions:         (dim_1: d1, dim_2: d2, dim_3: d3)
Coordinates:
    coord_1       (dim_1) data_type_coord_1 dask.array<chunksize=(chunk_d1,), meta=np.
↪ndarray>
    coord_2       (dim_1, dim_2)  data_type_coord_2 dask.array<chunksize=(chunk_d1,
↪chunk_d2), meta=np.ndarray>
  * dim_1         (dim_1) data_type_dim_1 np.array([x_1, x_2, ..., x_d1])
  * dim_2         (dim_2) data_type_dim_2 np.array([y_1, y_2, ..., y_d2])
  * dim_3         (dim_3) data_type_dim_3 np.array([z_1, z_2, ..., z_d3])

Data variables:
```

(continues on next page)

(continued from previous page)

```
    DATA_1        (dim2) data_type_DATA_1 dask.array<chunksize=(chunk_d2,), meta=np.
↪ndarray>
    DATA_2        (dim1) data_type_DATA_2 dask.array<chunksize=(chunk_d1,), meta=np.
↪ndarray>
    DATA_3        (dim1,dim2,dim3) data_type_DATA_3 dask.array<chunksize=(chunk_d1,
↪chunk_d2,chunk_d3), meta=np.ndarray>

Attributes:
    attr_1:         a1
    attr_2:         a2
```

- d1, d2, d3 are integers.
- a1, a2 can be any data type that can be stored in a JSON file.
- data_type_dim_, data_type_coord_ , data_type_DATA_ can be any acceptable NumPy array data type.

Explanations of dimension coordinates, coordinates and data variables from the Xarray website:

- dim_ "Dimension coordinates are one dimensional coordinates with a name equal to their sole dimension (marked by * when printing a dataset or data array). They are used for label based indexing and alignment, like the index found on a pandas DataFrame or Series. Indeed, these dimension coordinates use a pandas. Index internally to store their values."
- coord_ "Coordinates (non-dimension) are variables that contain coordinate data, but are not a dimension coordinate. They can be multidimensional (see Working with Multidimensional Coordinates), and there is no relationship between the name of a non-dimension coordinate and the name(s) of its dimension(s). Non-dimension coordinates can be useful for indexing or plotting; otherwise, xarray does not make any direct use of the values associated with them. They are not used for alignment or automatic indexing, nor are they required to match when doing arithmetic."
- DATA_ The array that dimension coordinates and coordinates label.

11.3.3 Zarr

Data is stored in Zarr files with Xarray formatting (which is specified in the metadata). The data is stored as N-dimensional arrays that can be chunked on any axis. The ngCASA/CNGI data formats are:

- vis.zarr Visibility data (measurement set).
- img.zarr Images (also used for convolution function cache).
- cal.zarr Calibration tables.
- tel.zar Telescope layout (used for simulations)
- Other formats will be added as needed.

CNGI will provide functions to convert between the new Zarr formats and legacy formats (such as the measurement set, FITS files, ASDM, etc.). The current implementations of the Zarr formats are not fully developed and are only sufficient for prototype development.

Zarr data formats rules:

- Data variable names are always uppercase and dimension coordinates are lowercase.
- Data variable names are not fixed but defaults exist, for example the data variable that contains the uvw data default name is UVW. This flexibility allows for the easy inclusion of more advanced algorithms (for example multi-term deconvolution that produces Taylor term images).
- Dimension coordinates and coordinates have fixed names.

- Dimension coordinates are not chunked.
- Data variables and coordinates are chunked and should have consistent chunking with each other.
- Any number of data variables can be in a dataset but must share a common coordinate and chunking system.

Zarr is the chosen data storage library for the CNGI Prototype. It provides an implementation of chunked, compressed, N-dimensional arrays that can be stored on disk, in memory, in cloud-based object storage services such as Amazon S3, or any other collection that supports the MutableMapping interface.

Compressed arrays can be hierarchically organized into labeled groups. For the CNGI Prototype, the Zarr group hierarchy is structured to be compatible with the Xarray dataset convention (a Zarr group contains all the data for an Xarray dataset):

```
My/Zarr/Group
    |-- .zattrs
    |-- .zgroup
    |-- .zmetadata

    |-- Array_1
    |    |-- .zattrs
    |    |-- .zarray
    |    |-- 0.0.0. ... 0
    |    |-- ...
    |    |-- C_1.C_2.C_3. ... C_D
    |-- ...
    |-- Array_N_c
    |    |-- ...
```

The group is accessed by a logical storage path (in this case, the string `My/Zarr/Group`) and is self-describing; it contains metadata in the form of hidden files (`.zattrs`, `.zgroup` and `.zmetadata`) and a collection of folders. Each folder contains the data for a single array along with two hidden metadata files (`.zattrs`, `.zarray`). The metadata files are used to create the lazily loaded representation of the dataset (only metadata is loaded). The data of each array is chunked and stored in the format `x_1,x_2,x_3, ..., x_D` where D is the number of dimensions and x_i is the chunk index for the ith dimension. For example a three dimensional array with two chunks in the first dimension and three chunks in the second dimension would consist of the following files `0.0.0, 1.0.0, 0.1.0, 1.1.0, 0.2.0, 1.2.0`.

Group folder metadata files (encoded using JSON):

- `.zgroup` contains an integer defining the version of the storage specification. For example:

```
{
   "zarr_format": 2
}
```

- `.zattrs` describes data attributes that can not be stored in an array (this file can be empty). For example:

```
{
"append_zarr_time": 1.8485729694366455,
"auto_correlations": 0,
"ddi": 0,
"freq_group": 0,
"ref_frequency": 372520022603.63745,
"total_bandwidth": 234366781.0546875
}
```

- `.zmetadata` contains all the metadata from all other metadata files (both in the group directory and array subdirectories). This file does not have to exist, however it can decrease the time it takes to create the

lazy loaded representation of the dataset, since each metadata file does not have to be opened and read separately. If any of the files are changed or files are added the `.zmetadata` file must be updated with `zarr.consolidate_metadata(group_folder_name)`.

Array folder metadata files (encoded using JSON):

- `.zarray` describes the data: how it is chunked, the compression used and array properties. For example the `.zarray` file for a DATA array (contains the visibility data) would contain:

```
{
"chunks": [270,210,12,1],
"compressor": {"blocksize": 0,"clevel": 2,"cname": "zstd","id": "blosc","shuffle":
↪ 0},
"dtype": "<c16",
"fill_value": null,
"filters": null,
"order": "C",
"shape": [270,210,384,1],
"zarr_format": 2
}
```

  Zarr supports all the compression algorithms implemented in numcodecs ('zstd', 'blosclz', 'lz4', 'lz4hc', 'zlib', 'snappy').

- `.zattrs` is used to label the arrays so that an Xarray dataset can be created. The labeling creates three types of arrays:
  - Dimension coordinates are one dimensional arrays that are used for label based indexing and alignment of data variable arrays. The array name is the same as its sole dimension. For example the `.zattrs` file in the "chan" array would contain:

```
{
"_ARRAY_DIMENSIONS": ["chan"]
}
```

  - Coordinates can have any number of dimensions and are a function of dimension coordinates. For example the "declination" coordinate is a function of the d0 and d1 dimension coordinates and its `.zattrs` file contains:

```
{
"_ARRAY_DIMENSIONS": ["d0","d1"]
}
```

  - Data variables contain the data that dimension coordinates and coordinates label. For example the DATA data variable's `.zattrs` file contains:

```
{
"_ARRAY_DIMENSIONS": ["time","baseline","chan","pol"],
"coordinates": "interval scan field state processor observation"
}
```

Zarr Advantages:

- Designed for concurrency, compatible out-of-the-box with chosen parallelism framework (Dask).
- Wide variety of compression algorithms supported, see numcodecs. Each data variable can be compressed using a different compression algorithm.
- Supports chunking along any dimension.
- Has a defined cloud interface.

11.3.4 Chunking

In the zarr and dask model, data is broken up into chunks to improve efficiency by reducing communication and increasing data locality. The zarr chunk size is specified at creation or when converting from CASA6 format datasets. The Dask chunk size can be specified in the `cngi.dio.read_vis` and `cngi.dio.read_image` calls using the `chunks` parameter. The dask and zarr chunking do not have to be the same. The zarr chunking is what is used on disk and the dask chunking is used during parallel computation. However, it is more efficient for the dask chunk size to be equal to or a multiple of the zarr chunk size (to stop multiple reads of the same data). This hierarchy of chunking allows for flexible and efficient algorithm development. For example cube imaging is more memory efficient if chunking is along the channel axis (the benchmarking example demonstrates this). Note, chunking can be done in any combination of dimensions.

11.3.5 Numba

Numba is an open source JIT (Just In Time) compiler that uses LLVMto translate a subset of Python and NumPy code into fast machine code from a chain of intermediate representations. Numba is used in ngCASA for functions that have long nested for loops (for example the gridder code). Numba can be used by adding the @jit decorator above a function:

```
@jit(nopython=True, cache=True, nogil=True)
def my_func(input_parms):
    does something ...
```

Explanation of jit arguments from the Numba:

`nopython`: > The behaviour of the nopython compilation mode is to essentially compile the decorated function so that it will run entirely without the involvement of the Python interpreter. This is the recommended and best-practice way to use the Numba jit decorator as it leads to the best performance.

`cache`: > To avoid compilation times each time you invoke a Python program, you can instruct Numba to write the result of function compilation into a file-based cache.

`nogil`: > Whenever Numba optimizes Python code to native code that only works on native types and variables (rather than Python objects), it is not necessary anymore to hold Python's global interpreter lock (GIL)"

A 5 minute guide to starting with Numba can be found here.

Numba also has functionality to run code on CUDA and AMD ROC) GPUs. This will be explored in the future.

11.3.6 Parallel Code with Dask

Code can be parallelized in three different ways:

- Built in Dask array functions. The list of dask.array functions can be found here. For example the fast Fourier transform is a built in parallel function:

```
uncorrected_dirty_image = dafft.fftshift(dafft.ifft2(dafft.ifftshift(grids_and_sum_
↪weights[0], axes=(0, 1)),
                                                    axes=(0,1)), axes=(0, 1))
```

- Apply a custom function to each Dask data chunk. There are numerous Dask functions, with varying capabilities, that do this: map_blocks, map_overlap, apply_gufunc, blockwise. For example the `dask.map_block` function is used to divide each image in a channel with the gridding convolutional kernel:

```
def correct_image(uncorrected_dirty_image, sum_weights, correcting_cgk):
    sum_weights[sum_weights == 0] = 1
    corrected_image = (uncorrected_dirty_image / sum_weights[None, None, :, :])
        / correcting_cgk[:, :, None, None]
    return corrected_image

corrected_dirty_image = dask.map_blocks(correct_image, uncorrected_dirty_image,
                                        grids_and_sum_weights[1],correcting_cgk_image)
```

- Custom parallel functions can be built using dask.delayed objects. Any function or object can be delayed. For example the gridder is implemented using dask.delayed:

```
for c_time, c_baseline, c_chan, c_pol in iter_chunks_indx:
    sub_grid_and_sum_weights = dask.delayed(_standard_grid_numpy_wrap)(
    vis_dataset[grid_parms["data_name"]].data.partitions[c_time, c_baseline, c_chan,␣
↪c_pol],
    vis_dataset[grid_parms["uvw_name"]].data.partitions[c_time, c_baseline, 0],
    vis_dataset[grid_parms["imaging_weight_name"]].data.partitions[c_time, c_baseline,
↪ c_chan, c_pol],
    freq_chan.partitions[c_chan],
    dask.delayed(cgk_1D), dask.delayed(grid_parms))
    grid_dtype = np.complex128
```

11.4 Documentation

All CNGI documentation is automatically rendered from files placed in the **docs** folder using the Sphinx tool. A Readthedocs service scans for updates to the Github repository and automatically calls Sphinx to build new documentation as necessary. The resulting documentation html is hosted by readthedocs as a CNGI website.

Compatible file types in the docs folder that can be rendered by Sphinx include:

- Markdown (.md)
- reStructuredText (.rst)
- Jupyter notebook (.ipynb)

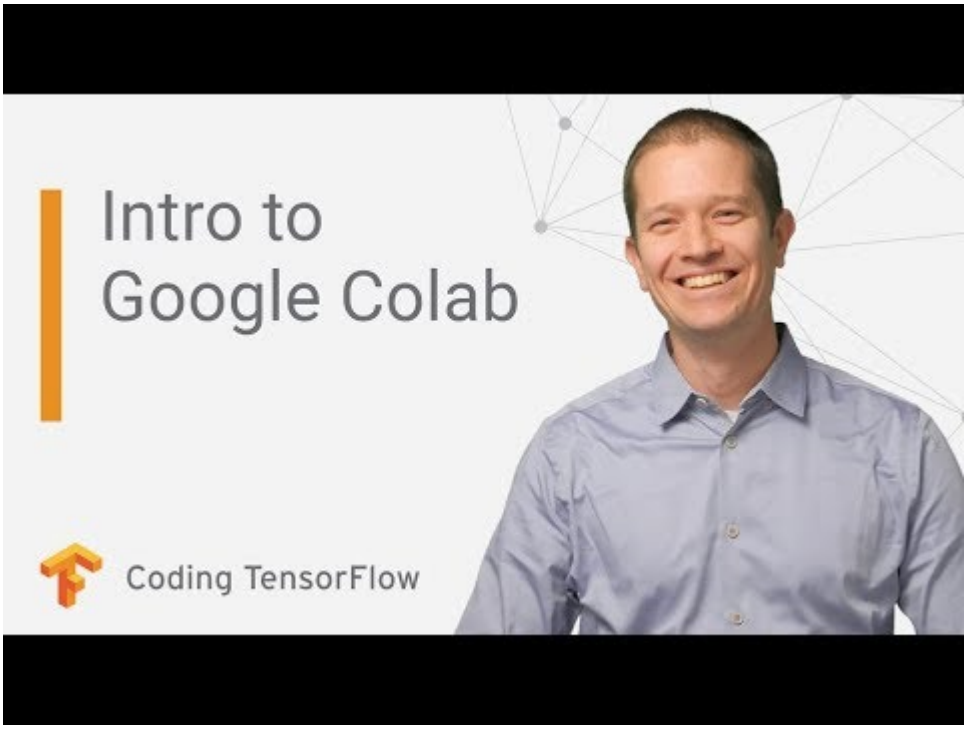
Sphinx extension modules are used to automatically crawl the cngi code directories and pull out function definitions. These definitions end up in the API section of the documentation. All CNGI functions must conform to the numpy docstring format.

The nbsphinx extension module is used to render Jupyter notebooks to html.

11.5 IDE

The CNGI team recommends the use of the PyCharm IDE for developing CNGI code. PyCharm provides a simple (relatively) unified environment that includes Github integration, code editor, python shell, system terminal, and venv setup.

CNGI also relies heavily on Google Colaboratory for both documentation and code execution examples. Google colab notebooks integrate with Github and allow markdown-style documentation interleaved with executable python code. Even in cases where no code is necessary, colab notebooks are the preferred choice for markdown documentation. This allows other team members to make documentation updates in a simple, direct manner.



11.6 PyPi Packages

CNGI is distributed and installed via pip by hosting packages on pypi. The pypi test server is available to all authorized CNGI developers to upload an evaluate their code branches.

Typically, the Colab notebook documentation and examples will need a pip installation of CNGI to draw upon. The pypi test server allows notebook documentation to temporarily draw from development branches until everything is finalized in a Github pull request and production pypi distribution.

Developers should create a .pypirc file in their home directory for convenient uploading of distributions to the pip test server. It should look something like:

```
[distutils]
index-servers =
     pypi
     pypitest

[pypi]
username = yourusername
password = yourpassword

[pypitest]
repository = https://test.pypi.org/legacy/
username = yourusername
password = yourpassword
```

Production packages are uploaded to the main pypi server by a subset of authorized CNGI developers when a particular version is ready for distribution.

11.7 Step by Step

Concise steps for contributing code to CNGI

11.7.1 Install IDE

1. Request that your Github account be added to the contributors of the CNGI repository
2. Make sure Python 3.6 and Git are installed on your machine
3. Download and install the free PyCharm Community edition. On Linux, it is just a tar file. Expand it and execute pycharm.sh in the bin folder via something like:

```
$ ./pycharm-community-2020.1/bin/pycharm.sh
```

4. From the welcome screen, click `Get from Version Control`
5. Add your Github account credentials to PyCharm and then you should see a list of all repositories you have access to
6. Select the CNGI repository and set an appropriate folder location/name. Click "Clone".
7. Go to:

```
File -> Settings -> Project: xyz -> Python Intrepreter
```

   and click the little cog to add a new Project Interpreter. Make a new **Virtualenv** environment, with the location set to a venv subfolder in the project directory. Make sure to use Python 3.6.
8. Double click the `requirements.txt` file that was part of the git clone to open it in the editor. That should prompt PyCharm to ask you if you want to "Install requirements" found in this file. Yes, you do. You can ignore the stuff about plugins.
9. All necessary supporting Python libraries will now be installed in to the venv created for this project (isolating them from your base system). Do NOT add any project settings to Git.

11.7.2 Develop stuff

1. Double click on files to open in editor and make changes.
2. Create new files with:

```
right-click -> New
```

3. Move / rename / delete files with:

```
right-click -> Refactor
```

4. Run code interactively by selecting "Python Console" from the bottom of the screen. This is your venv enviornment with everything from requirements.txt installed in addition to the cngi package. You can do things like this:

```
>>> from cngi.dio import read_vis
>>> xds = read_vis('path\to\data.vis.zarr')
```

5. When you make changes to a module (lets say read_vis for example), close the Python Console and re-open it, then import the module again to see the changes.
6. Commit changes to your local branch with

```
right-click -> Git -> Commit File
```

7. Merge latest version of Github master trunk to your local branch with

```
right-click -> Git -> Repository -> Pull
```

8. Push your local branch up to the Github master trunk with

```
right-click -> Git -> Repository -> Push
```

11.7.3 Make a Pip Package

1. If not already done, create an account on pip (and the test server) and have a CNGI team member grant access to the package. Then create a `.pypirc` file in your home directory.
2. Set a unique version number in `setup.py` by using the release candidate label, as in:

```
version='0.0.48rc1'
```

3. Build the source distribution by executing the following commands in the PyCharm Terminal (button at the bottom left):

```
$ rm -fr dist
$ python setup.py sdist
```

4. call twine to upload the sdist package to pypi-test:

```
$ python -m twine upload dist/* -r pypitest
```

5. Enjoy your pip package as you would a real production one by pointing to the test server:

```
$ pip install --extra-index-url https://test.pypi.org/simple/ cngi-prototype==0.0.
↪48rc1
```

11.7.4 Update the Documentation

1. A bulk of the documentation is in the `docs` folder and in the '.ipynb' format. These files are visible through PyCharm, but should be edited and saved in **Google Colab**. The easiest way to do this is not navigate to the Github docs folder and click on the .ipynb file you want to edit. There is usually an `open in colab` link at the top.
2. Alternatively, notebooks can be accessed in Colab by combining a link prefix with the name of the .ipynb file in the repository `docs` folder. For example, this page you are reading now can be edited by combining the colab prefix:

```
https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/
```

with the filename of this notebook:

```
development.ipynb
```

producing a link of: https://colab.research.google.com/github/casangi/cngi_prototype/blob/master/docs/development.ipynb

3. In Colab, make the desired changes and then select

```
File -> Save a copy in Github
```

enter you Github credentials if not already stored with Google, and then select the CNGI repository and the appropriate path/filename, i.e. `docs/development.ipynb`

4. Readthedocs will detect changes to the Github master and automatically rebuild the documentation hosted on their server (this page you are reading now, for example). This can take ~15 minutes

In the docs folder, some of the root index files are stored as .md or .rst format and may be edited by double clicking and modifying in the PyCharm editor. They can then be pushed to the master trunk in the same manner as source code.

After modifying an .md or .rst file, double check that it renders correctly by executing the following commands in the PyCharm Terminal

```
$ cd docs/
$ rm -fr _api/api
$ rm -fr build
$ sphinx-build -b html . ./build
```

Then open up a web browser and navigate to

```
file:///path/to/project/docs/build/index.html
```

Do **NOT** add api or build folders to Git, they are intermediate build artifacts. Note that **_api** is the location of actual documentation files that automatically parse the docstrings in the sourcecode, so that *should* be in Git.

11.8 Coding Standards

Documentation is generated using Sphinx, with the autodoc and napoleon extensions enabled. Function docstrings should be written in NumPy style. For compatibility with Sphinx, import statements should generally be underneath function definitions, not at the top of the file.

A complete set of formal and enforced coding standards have not yet been formally adopted. Some alternatives under consideration are:

- Google's style guide
- Python Software Foundation's style guide
- Following conventions established by PyData projects (examples one and two)

We are evaluating the adoption of PEP 484 convention, mypy, or param for type-checking, and flake8 or pylint for enforcement.

CHAPTER

TWELVE

ABOUT THIS PROJECT

This is a rapid prototype development effort underway by members of the Common Astronomy Software Applications (CASA) team at the National Radio Astronomy Observatory (NRAO).

https://casa.nrao.edu

The CASA team has begun exploring options for a new generation of software to meet the growing demands of current and future instruments. This **cngi_prototype** package is a demonstration of the current state of our research efforts. Its primary purpose is to showcase new data structures for MeasurementSet and Image contents built entirely in Python atop the popular technology stack of numpy, dask, and xarray. A selection of core mathematics, manipulation, middleware and analysis functions are shown to demonstrate the simplicity and scalability of the technology choices. Finally, the most compute intensive areas of CASA imaging are implemented and benchmarked to demonstrate the parallel scalability and raw performance now possible from a pure-Python software stack.

This software is subject to change without notice. It has not been thoroughly tested or verified and should be considered demonstration only. An official release under a different name is planned for the future, at which time more rigorous testing and change control/communication will be in place.

Questions and feedback can be sent to: **casa-feedback@nrao.edu**

PYTHON MODULE INDEX

c

n

INDEX

D

E

F

G

I

J

L

M

N

P

Q

R

S

T

U

V

W